Causal Weak-Consistency Replication – A Systems Approach

DISSERTATION

zur Erlangung des akademischen Grades
doctor rerum naturalium
(Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt-Universität zu Berlin

von
Herr Diplom Informatiker Felix Hupfeld
geboren am 27.9.1976 in Ulm

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Christoph Markschies

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II:
Prof. Dr. Wolfgang Coy

Gutachter:

1. Prof. Dr. Alexander Reinefeld
2. Prof. Dr. Jens-Peter Redlich
3. Prof. Dr. Marc Shapiro

Tag der Verteidigung: 28. Januar 2009

Abstract

Data replication techniques introduce redundancy into a distributed system architecture that can help solve several of its persistent problems. In wide area or mobile systems, a replication system must be able to deal with the presence of unreliable, high-latency links. Only asynchronous replication algorithms with weak-consistency guarantees can be deployed in these environments, as these algorithms decouple the local acceptance of changes to the replicated data from coordination with remote replicas.

This dissertation proposes a framework for building weak-consistency replication systems that provides the application developer with causal consistency guarantees and mechanisms for handling concurrency. By presenting an integrated set of mechanisms, algorithms and protocols for capturing and disseminating changes to the replicated data, we show that causal consistency and concurrency handling can be implemented in an efficient and versatile manner. The framework is founded on log of changes, which both acts the core data structure for its distributed algorithms and protocols and serves as the database log that ensures the consistency of the local data replica.

The causal consistency guarantees are complemented with two distributed algorithms that handle concurrent operations. Both algorithms are based on the observation that uncoordinated concurrent operations introduce a divergence of state in a replication system that can be modeled as the creation of version branches. Distributed Consistent Branching (DCB) recreates these branches on all participating processes in a consistent manner. Distributed Consistent Cutting (DCC) selects one of the possible branches in a consistent and application-controllable manner and enforces a total causal order for all its operations.

The contributed algorithms and protocols were validated in an database system implementation, and several experiments assess the behavior of these algorithms and protocols under varying conditions.

Keywords:
weak-consistency replication, optimistic replication, eventual consistency, gossip

Zusammenfassung

Replikation kann helfen, in einem verteilten System die Fehlertoleranz und Datensicherheit zu verbessern. In Systemen, die über Weitverkehrsnetze kommunizieren oder mobile Endgeräte einschließen, muß das Replikationssystem mit großen Kommunikationslatenzen umgehen können. Deshalb werden in solchen Systemen in der Regel nur asynchrone Replikationsalgorithmen mit schwach-konsistenter Änderungssemantik eingesetzt, da diese die lokale Annahme von Änderungen der Daten und deren Koordinierung mit anderen Replikaten entkoppeln und somit ein schnelles Antwortverhalten bieten können.

Diese Dissertation stellt einen Ansatz für die Entwicklung schwach-konsistenter Replikationssysteme mit erweiterten kausalen Konsistenzgarantien vor und weist nach, daß auf seiner Grundlage effiziente Replikationssysteme konstruiert werden können. Dazu werden Mechanismen, Algorithmen und Protokolle vorgestellt, die Änderungen an replizierten Daten aufzeichnen und verteilen und dabei Kausalitätsbeziehungen erhalten. Kern ist ein Änderungsprotokoll, das sowohl als grundlegende Datenstruktur der verteilten Algorithmen agiert, als auch für die Konsistenz der lokalen Daten nach Systemabstürzen sorgt.

Die kausalen Garantien werden mit Hilfe von zwei Algorithmen erweitet, die gleichzeitige Änderungen konsistent handhaben. Beide Algorithmen basieren auf der Beobachtung, daß die Divergenz der Replikate durch unkoordinierte, gleichzeitige Änderungen nicht unbedingt als Inkonsistenz gesehen werden muß, sondern auch als das Erzeugen verschiedener Versionen der Daten modelliert werden kann. Distributed Consistent Branching (DCB) erzeugt diese alternativen Versionen der Daten konsistent auf allen Replikaten; Distributed Consistent Cutting (DCC) wählt eine der Versionen konsistent aus.

Die vorgestellten Algorithmen und Protokolle wurden in einer Datenbankimplementierung validiert. Mehrere Experimente zeigen ihre Einsetzbarkeit und helfen, ihr Verhalten unter verschiedenen Bedingungen einzuschätzen.

Schlagwörter:
Schwach-konsistente Replikation, Optimistische Replikation, eventual consistency, Gossip

Contents

Chapter 1

Introduction

Together with file systems, the storage capabilities of databases and other structured storage systems are among the most important persistence mechanisms for applications. Augmenting these mechanisms with replication can help solve several challenging design problems: a replication facility can ensure the availability and safety of data and improve service performance by enabling parallel access. The work in this thesis focuses on weak-consistency replication techniques, which decouple data accesses from their coordination over the network and allow storage systems to better support uncontrolled environments such as large data centers, wide area or mobile networks.

Applying replication to structured storage systems is still a challenge, and no general solution has been established [19, 11]. An engineer faced with this task can not directly revert to the existing body of research in this area, because it is non-trivial to transform the mostly formal descriptions of replication algorithms from the research literature into an implementation [22]. Furthermore, such a transformation can have considerable performance implications that cannot be easily inferred from reading a formal description of an algorithm. Consequently, today's production implementations lag behind the state of the art in research and are usually hand-crafted to fit a particular problem.

This dissertation contributes a solution to the problem of designing and implementing real-world weak-consistency replication systems for structured data. We assert that causal consistency is a well-qualified foundation for such a solution and show that a combination of causality-preserving protocols and persistent logs makes it possible to craft a comprehensive framework of the core components of a weak-consistency replication system. This framework integrates algorithms, mechanisms and protocols that exploit existing system primitives to yield efficient and lightweight replication systems and can be applied to various replication problems.

1.1 Weak-Consistency Replication and its Applications

Until recently weak-consistency replication algorithms occupied a relatively small niche in the design space for replicated storage. This was largely due to the fact

that users expected databases and other structured storage systems to provide strong consistency guarantees in the form of a tightly-integrated system component. These expectations have been challenged by leading system researchers [110, 107, 50, 4, 1, 57] for some time and their arguments are now gaining acceptance as a wider audience understands that the status quo of structured storage cannot be maintained in the face of new challenges posed by large data centers and mobile networks.

In these environments, where network communication is no longer a reliable commodity, only weak-consistency replication is feasible as it does not rely on synchronous coordination of changes. The benefits of this asynchrony are manifold: it leads to a loose coupling between systems, with the effect that failures and temporary performance problems of single nodes do not influence the overall system; and it allows scaling to much larger systems, as coordination can be carried out lazily.

The loose coupling and fault-tolerance of weak-consistency replication are especially important in **large-scale computing infrastructures** that have been scaled to tens or hundreds of thousands of computers with the help of distributed storage systems such as BigTable [21] (at Google) and Dynamo [32] (at Amazon). These systems dispense with the relational data model and strong transactional guarantees, yet fault tolerance is still essential to the operation of their respective companies, whose multi-billion dollar quarterly revenues are solely generated via their IT infrastructures and therefore any outage directly translates into considerable sums of lost money.

Weak-consistency replication techniques are also essential for systems that operate in **challenged networks**, a class of network environments that subsumes mobile terrestrial, mobile ad-hoc, sensor and other kinds of networks that can experience high packet loss rates, long communication latency and frequent or steady network partitions [39]. In these environments, the ability of weak-consistency replication is important to disseminate changes as communication is available and still eventually reach consistency of the replicated data.

1.2 Towards General Weak-Consistency Replication

The primary contribution of this dissertation is a general framework for building weak-consistency storage systems that is not tied to any particular application or system and can be applied to a wide class of applications.

The task of such a general weak-consistency replication framework is to provide abstractions, components and interfaces that solve the most important challenges of building a weak-consistency replication system in a reusable way. In particular, the framework must include a complete stack of mechanisms for interacting with the application, persisting data, disseminating changes and keeping them consistent, and managing storage resources. The design should make the required system primitives explicit and should be transparent in a way that facilitates performance analysis and prediction for a specific problem. The design of such a general architecture requires careful consideration of several theoretical and practical aspects:

Programming model. The choice of a programming model is caught in the tension between simplicity of the programming interface and the consistency model it implies and which ultimately determines the performance and operation of the overall system. The closer the model resembles the single-copy consistency model and programming interface of normal databases, the lower the entry barrier for programmers. However, if the model mimics existing interfaces too closely, a weak-consistency application cannot play out its advantages and needs expensive protocols to provide its semantical guarantees. For example, the provision of a POSIX interface to files implies sequential consistency guarantees that requires expensive strong-consistency replication algorithms [115].

Consequently, the challenge is to design an interface that is simple for programmers to grasp but that is not tied to a particular application or system. Because interface semantics are directly related to later system operation, the interface should not attempt to make its interface completely transparent, but provide explicit affordances that suggest how interface usage influences system operation. This allows the application programmer to adapt its design and implementation to the laws of distributed systems and consciously pay for decisions with performance. It also avoids creating an unintended "semantical safety-margin", which can happen if the programmer is not conscious of the effects of his choices.

Non-functional theoretical properties. The consistency semantics that the replication system provides directly determine the theoretical boundaries for fault-tolerance and performance that the replication system will be able to achieve [45]. In order to be able to exploit its conceptual advantages, a general weak-consistency replication architecture should be based on loosely-coupled algorithms and protocols. In particular, a weak-consistency replication system can only play out is unique advantages if its design allows it operate in presence of failures of many replicas, large communication latencies and network partitions.

Performance and complexity of the implementation. When abstractions in the algorithm design do not consider implementation concerns, the quality of an implementation can suffer dramatically. In particular, abstractions that ignore the mechanics of existing hardware can perform poorly when implemented. A good example for this is the Paxos algorithm, which on first sight appears as ready-to-use consensus algorithm, but whose inefficient use of permanent storage does not allow its direct implementation [20]. Similarly, when primitives are chosen from a too-high level of abstractions, the implementation of the full system can grow complex, which can in turn affect the runtime performance of the implementation [22].

1.3 Approach and Contributions

The contributed weak-consistency replication framework is the result of a **deductive systems-oriented bottom-up process**: instead of starting from a specification

that is derived from postulated use cases, we build on the concept of causality and extend it to provide abstractions that serve relevant classes of applications in an efficient manner. This process leads us from existing system primitives to abstractions that match these primitives (and are therefore efficient and light-weight) while representing valuable higher-level mechanisms for applications. Because we apply this process to all levels of our design, it also results directly in a programming model that does not hide the mechanics of the system and therefore allows predictions of performance of design choices.

We propose to build a replication system entirely around augmented causal consistency guarantees and explore the limits of this approach. The concept of **causality** is an abstraction of the important property of a distributed computing system that any computation is potentially dependent on all information that is available to the computation at its execution time. This dependency does not only include all the results of previous computation steps, but also information that has reached the computing process from remote processes over network links.

With our contributions we show that causality and causal consistency are a good foundation for a weak-consistency replication framework. The contributions provide all mandatory components of a framework for implementing weak-consistency replication systems:

Programming model and concurrency handling. Causal consistency results in a convenient programming model because it provides what a programmer is used to: it respects data dependencies and any read operation reflect the result of all preceding write operations. We show that causal consistency can be made useful for applications by completing it with consistency algorithms that handle concurrent changes. We observe that concurrent changes can be modeled as the implicit creation of separate version **branches** of the data and develop two algorithms that handle concurrent changes in a way that leads to eventually consistent replicas. **Distributed Consistent Branching** for mobile systems handles concurrent changes by explicitly creating version branches on all replicas in a consistent manner. **Distributed Consistent Cutting** for server replication handles concurrent changes by consistently selecting one branch of concurrent operations over the others and providing total causal order for this one branch.

Theoretical properties. We show that causal consistency enables non-blocking replication system architectures, which translates directly into a loosely-coupled system architecture. To that end, we rely on an operation model that records the effect of changes on the data instead of merely recording the operations that cause the change. Changes recorded this way are disseminated by two causality-tracking communication protocols: the **causal gossip** protocol (for mobile and other challenged networks) efficiently extends gossip-based communication to preserve and convey causal dependencies between changes in a highly efficient manner. **Direct send** (for online replication of servers) is an augmented reliable FIFO multicast proto-

col for which we elicit important properties that make FIFO delivery applicable to causal consistency.

Implementation and systems aspects. Causality-preserving communication and persistence mechanisms are a natural extension to existing system primitives for network and disk I/O and thus facilitate an efficient and lightweight implementation. Our two protocols causal gossip and direct send can directly use operating system primitives for unreliable message communication. Furthermore, all our algorithms are designed in a way so that make them use storage as a log in an append-only manner. Log storage is known to be an **efficient structure for persistence** whose architectural properties are well understood [120] and have been advocated for use in databases [80] and file systems [98]. We use the log's sequential storage to build a log-structured persistence mechanism that is able to efficiently record and preserve causal relationships across processes and thereby avoid other persistent representations of causality such as version vectors. Apart from capturing causal relationships in a node, we show how the replication log can be unified with the **database redo log** that guarantees local consistency and serves as a crash recovery log, saving storage space and disk bandwidth. We also provide the necessary mechanisms for log storage management and cleanup.

1.4 Thesis Outline

We continue this thesis with chapter 2 – *Terms and Concepts – Related Work*, which gives an introduction to basic terms and concepts of the field of distributed algorithms and provides an overview of research literature on weak-consistency replication. Chapter 3 – *System Model and Assumptions* describes the assumptions that we made about the environment in which our replication algorithms shall operate. In chapter 4 – *Tracking and Disseminating Changes*, we describe how we capture changes to the replicated data and how causal gossip and direct send record and disseminate changes. In chapter 5 – *The Log as a Storage System* we highlight several important aspects of using a persistent log as part of a structured storage system. In chapter 6 – *Maintaining Consistency without Coordination* we describe the Distributed Consistent Branching and Distributed Consistent Cutting algorithms that handle concurrent operations. The last chapter, chapter 7 – *Evaluation* describes the experiments that we have performed with the implementation of our replicated database system. Chapter 8 - *Conclusion* summarizes the findings of this thesis.

Chapter 2

Terms and Concepts – Related Work

This chapter introduces the reader to basic concepts in distributed systems and reviews the literature that has been published on replication system architectures.

2.1 Basic Concepts of Distributed Systems

In order to be able to reason about distributed systems and their properties independent from the actual hardware they run on, the nodes that participate in the system are modeled as a finite **set of processes** $p_i \in P$. These processes perform computations and communicate by exchanging **messages** over communication **links**[1].

2.1.1 Processes and Events

The process abstraction of a distributed system is founded in the abstract model of a **computing machine**. This model abstracts from a real computer by subsuming the contents of its volatile and persistent memory as the abstract **state** of the machine. Starting from an initial state, the machine transforms its state in discrete steps by the serial execution of an algorithm.

The **process** abstraction extends this model of a computing machine to distributed systems. A process does not only perform computations but is also able to communicate with other processes by sending and receiving messages.

The set of states the process can assume is called its **state space**. The states of a process p can be formally represented as its **state space** St_p. The process' state is not only affected by computations, but also by sending and receiving messages (see Fig. 2.1). When any of these three potentially state-changing operation occurs, we say that an **event** has happened in the process.

Because of serial execution, only one event can happen at a time and thus all events that happen in a process are totally ordered. Consequently, the series of states from this state space that a process goes through during its lifetime can be identified by the timestamps of a logical clock:

[1] See for example [53] or any other text book on distributed systems.

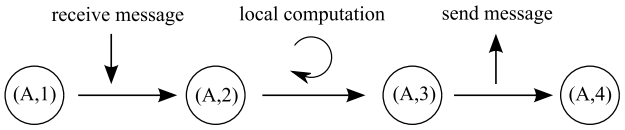


Figure 2.1: State transitions of a process

Definition 2.1 *A* logical clock T_p *of a process* p *is a function* $T_p : \mathbb{N} \rightarrow St_p$. *A* logical timestamp t *of the logical clock* T_p *of a process* p *is a value of the logical clock:* $t \in \mathbb{N}$.

Because each event is causing a state transition, events can be unambiguously identified by the logical clock of the state that it caused the process to change over to. Therefore we can define:

Definition 2.2 *Let* e *be an event from the process* p *that caused the process to change over to the state* s *identified by* $(t,s) \in T_p$. *We define:*

- *the* event identifier $id(e)$ *as the tuple*
  $id(e) := (p,t)$,
- *the* logical timestamp $time(e)$ *as the logical timestamp of* p *after the event has happened:*
  $time(e) := t$,
- *the* event source $process(e)$ *as the process* p *at which the event happened,*
  $process(e) := p$.

As introduced, the abstractions of events and logical time capture the operation of a process on the lowest level and describe the physical reception and delivery of messages. This model however can also be used to describe algorithms that implement *higher-level abstractions*, for example a communication protocol for reliable communication. When higher-level abstractions are modeled with the process abstraction, the act of sending and receiving messages refers to messages with enhanced qualities (such as reliable messages) and the process abstraction hides the internal computation and message exchanges that are necessary to implement the abstraction. Also the physical reception of a message is separated from its **delivery** to upper layers for further processing.

Using this common model of a process that performs computation and communication, algorithms can be stacked on top of each other, resulting in a **layered architecture** with an increasing quality of abstraction. Each protocol or other form of distributed algorithm in the stack may consume messages for internal use, or delay the delivery to its higher-level clients.

2.1.2 System Model, Links, Protocols and Failure Detectors

Links are an abstraction of the communication medium that transfers messages between processes. The links of a distributed system model are not perfectly reliable communication media, as they are used to abstract from real physical communication facilities. For instance, in their most general form links can corrupt, drop, reorder, duplicate, or generate messages.

Links do not have to exist between every pair of processes (which reflects a situation where two processes cannot communicate directly with each other but only via a third process) and can be even asynchronous and allow message transfer in only one direction. Taken together, the links between processes form a directed graph that represents the network **topology** (Fig. 2.2). This topology may change over time and can also consist of disconnected subgraphs (*partitioned network*).

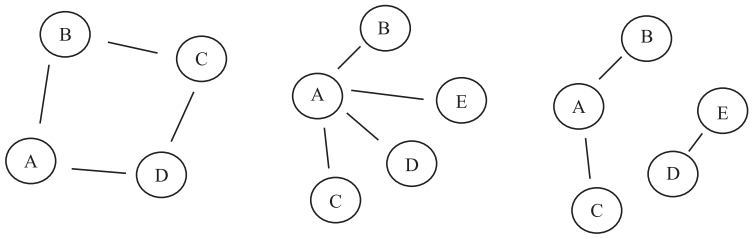


Figure 2.2: Three network topologies: ring, star, disconnected sub-networks

If a reliable communication facility is necessary, the deficiencies of a link can be mitigated with the help of a **communication protocol**. Communication protocols are distributed algorithms that use special messages to transparently simulate a link with stricter guarantees or a different topology. For instance, the TCP protocol uses acknowledgment messages, checksums and sequence numbers to hide loss, corruption and duplication of packets. The IP protocol, together with a routing protocol, allows hosts that are connected to different local area networks (LANs) to address each other directly [109].

While it is possible to improve the qualities of a link in a transparent manner through protocols, it is not always advisable to do so. Some applications are able to deal with deficiencies of the network in a more efficient, application-specific way, and the costs and behavior of general communication protocols would be even prohibitive for them.

A further important part of a distributed system model are **failure detectors** that introduce the notion of time to a distributed system. Failure detectors can be used in distributed algorithms to make progress even in the presence of failures by detecting any failures with the help of timeouts. Because the algorithms in this thesis are purely reactive and cannot block, time and failure detectors do not play a role in the algorithms of this dissertation.

2.1.3 Message Delivery and Ordering

Protocols can not only be used to enhance the quality of a link, but can also be applied to exercise control over the order in which messages are delivered to a process (**order of delivery**).

These ordering guarantees can be implemented in the source processes (**source ordering**) by controling in which order the messages are sent to the target or in the destination process (**destination ordering**) by delivering physically received messages to higher layers in the protocol stack only when the ordering constraint is satisfied.

Two important protocols for message ordering are First In First Out (FIFO) order delivery and total order delivery [33], which ensure that:

- **FIFO order delivery**. Messages that originate from the same process are delivered in the order they were sent. Messages from different process can be delivered in any order.

- **Total order delivery**. All messages are delivered in the same order to all processes.

2.2 The Concept of Causality in Distributed Systems

The dynamics of a distributed system can be visualized in a **process-time diagram** (Fig. 2.3). This diagram contains a time line for each process along with its events and any messages that have been sent between processes. For example, in figure 2.3 process A has one send event $(A, 1)$ that results in a receive event $(B, 2)$, and B performs a local computations at $(B, 1)$ and $(B, 3)$ and sends a message to C at $(B, 4)$.

The process-time diagram gives a global perspective on the distributed system that is helpful for visualizing the dynamics and communication of a set of distributed processes. However, this global view on the distributed system *is not available to any of the processes* and can not be used in distributed algorithms.

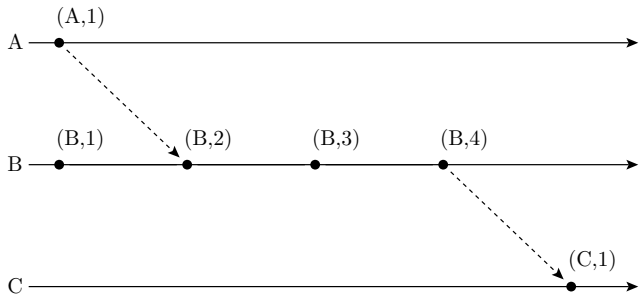


Figure 2.3: Example of a process-time diagram

2.2.1 The Causality Relation

In a process-time diagram it becomes evident that processes can potentially influence each other by communication. In figure 2.3, B's state at $(B,3)$ may have been influenced by events that happened at A because it received a message from A at $(B,2)$. For the same reason C may have been influenced by B and A after $(C,1)$. More generally, if a process receives a remote message, all its further actions are potentially influenced by the events that happened at the sending process up to the time the message was sent.

If an event was potentially influenced by a remote event, we say that the receiving process' state is **causally dependent** on the sending process' state [74, 75]. Of course, a process' state is also causally dependent on all the local events that have happened. We call all the events that a particular event is causally dependent on and which are therefore in its event history its **causal predecessors**.

The **is-causally-dependent-on** relationship is an **order relation** between two events of a distributed system. This relation is transitive, because an event potentially also "knows" everything that its causal predecessors know. It is also antisymmetric, because no two events can mutually influence each other. However, it is only a partial order because two events might not be causally related in either direction.

The causal dependencies of an event are defined over the messages that a process has received at the time when the event e was generated. The concept of causality describes a fact of distributed systems: when a process receives a message, its further actions are potentially influenced by that message and therefore by the sender of the message.

Definition 2.3 *An event e_2 is* directly causally dependent *on e_1 ($e_2 \xrightarrow{1} e_1$) if $process(e_1)$ sends a message to $process(e_2)$ after e_1 has happened and $process(e_2)$ receives this message before e_2 happens, or $process(e_1) = process(e_2)$ and $time(e_2) > time(e_1)$. An event e_2 is* causally dependent *on e_1 ($e_2 \longrightarrow e_1$) if there exist events e with $e_2 \xrightarrow{1} \ldots e \ldots \xrightarrow{1} e_1$ (transitive extension). An event e_1 is* causally related *with e_2, if $e_1 \longrightarrow e_2$ or $e_2 \longrightarrow e_1$.*

Events that are not causally related are said to have happened concurrently. From the causal dependency relation $\longrightarrow$ we can directly derive a definition of causal concurrency:

Definition 2.4 *An event e_1 is* concurrent *with e_2, $e_1 \parallel e_2$, if neither $e_1 \longrightarrow e_2$ nor $e_2 \longrightarrow e_1$ holds.*

This definition of concurrency leads to the concurrency relation, **is-concurrent-to**. Because of its definition, concurrency is symmetric. However, it is not transitive, because two causally related events can each be concurrent to a third event.

The **is-causally-dependent-on** relation has been originally introduced by Lamport as the (inverse) **happened-before** relation [74]. Although its name suggests

a relationships between events in the real time domain, it also refers to the logical time domain.

On first sight, causal order seems to be intuitively similar to the **real-time order** of events as it is conveyed by the real time timestamps of events. However, it differs substantially in two ways:

1. Real time clocks of processes can not be perfectly synchronized and therefore always have certain clock skew that describe the clock's difference to an imagined global time. Consequently, two events from different processes that are causally related in one direction may have real-time timestamps that suggest a causal relation in the other direction.

2. Causal influence needs communication between processes. When one event from one process happens significantly before an event from a second process (say a couple of minutes), it can happen that the events have no causal relationship because communication between the process was intermittent or slow.

2.2.2 Representations of Causality

Causal dependencies (as a partial order) can be represented with various structures. The first type of representation that is relevant in this dissertation uses a graph: with the help of the causal relation, we can visualize everything a particular process currently knows. If we interpret the events as nodes of a graph and the direct causal dependencies as edges, we can build a **causality graph** (sometimes called antecedence graph [37]). A causality graph built for a particular process represents the process's *current local view of the global system state.*

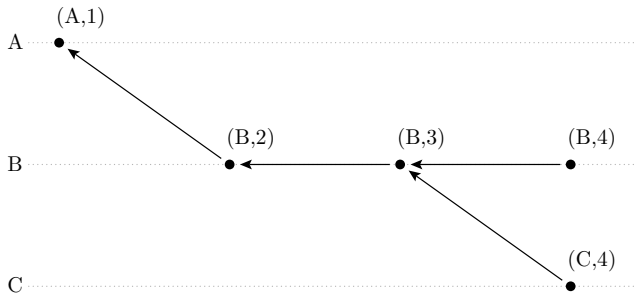


Figure 2.4: Example of a causality graph

The causality graph is a directed acyclic graph (DAG). A causal relation between two events exists if there is a path through the graph that connects their respective edges; the absence of such a path means concurrency of the two events. While the graph representation is well usable for humans, it is not readily accessible to algorithms: graph representations are not easy to store and the path search necessary for finding out the causal relation between two events is expensive.

A representation of causality that is more amenable for computation is using vectors of process timestamps, so-called **timestamp vectors** or **version vectors** [30].

Definition 2.5 *A timestamp vector v is a mapping $v : P \to \mathbb{N}$ that assigns each process from the set of processes P a timestamp.*

A timestamp vector summarizes the causal history of an event by retaining the logical timestamp of each of the event's direct causal predecessors. Because the causal relation is transitive, knowledge of the latest event implies knowledge of all its predecessors. Extended with a component-wise order relation that is satisfied when all of vector's components are larger than the components of the other vector, version vectors represent the partial order of causality.

For instance, the timestamp vector that represents the causal dependencies of event from Fig. 2.4 looks as follows when process A, B and C are assigned to the first, second and third vector component, respectively:

$$dep_{(B,2)} = \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}, \quad dep_{(B,4)} = \begin{pmatrix} 1 \\ 4 \\ 0 \end{pmatrix}, \quad dep_{(C,4)} = \begin{pmatrix} 1 \\ 3 \\ 4 \end{pmatrix}$$

Component-wise comparison results in the following causal relations:

$$dep_{(C,4)} \longrightarrow dep_{(B,2)}, \quad dep_{(B,4)} \longrightarrow dep_{(B,2)}, \quad dep_{(C,4)} \parallel dep_{(B,4)}$$

Although version vectors are well suited for computing causal relationships, storing them can be problematic in some application contexts, such as database replication [36]. A version vector can be relatively heavyweight, and a goal of research in this area has been [61, 81] to reduce their storage space by various means, usually by exploiting structures in the causal relationships [43, 87, 97]. The algorithms in this thesis avoid this drawback of timestamp vectors as they only operate on timestamp vectors as a temporary representation of causality and use otherwise the operation log for persisting them.

Besides version vectors and causality graphs, the causal relation of events can also be represented in several other ways [106, 94], such as event histories [92, 96], hash histories [70], version stamps [6], and concurrent regions [106].

2.2.3 Matrix Clocks

Causal relations between events do not only describe the knowledge an event had about another event, but they can also be used to capture the knowledge that a particular process has about other processes in the system. When a process receives a message which represents a certain causal dependency, the process implicitly gains partial knowledge about the overall system state in the sense of which events have reached which processes.

By gathering the latest causal dependencies that a process is made aware of through the reception of messages from other nodes, we can build a **matrix clock** at each process p. A matrix clock M_p of the process p is a matrix of column vectors, each of which represents p's knowledge about a what a process p_j knows [99]:

Definition 2.6 *A matrix clock M_p of a process p is a mapping $M_p : P \times P \rightarrow \mathbb{N}$ that assigns each pair (p_i, p_j) from the set of processes P the timestamp of the latest event that p_j has received from p_i as known by p.*

A matrix clock M_p of the process p is a matrix of column vectors, each of which represents p's knowledge about a what a process p_j knows. A matrix clock can be maintained by each of the processes in the system using information conveyed by the causal dependencies of remote events as they are received. The column vector of p_j can be updated with a new causal dependency vector when we receive a new event from p_j.

$$M_p = \begin{pmatrix} p_{1j-1} & p_{1j} & p_{1j+1} \\ p_{2j-1} & p_{2j} & p_{2j+1} \\ .. & p_{3j} & .. \\ .. & .. & .. \end{pmatrix}$$

In our example of Fig. 2.4, the knowledge about the overall system of the process that knows the depicted causal relations would be:

$$M_p = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 4 & 3 \\ 0 & 0 & 4 \end{pmatrix}$$

Although the matrix is a representation of the overall system state, it does not represent the current system state, but only a more or less outdated view as observed by one process. Hence, a matrix clock represents a process' local knowledge about the global state of the system.

Consequently, the column vectors can be interpreted as a local view on what other processes know, i.e. which events a particular remote process has already received from other processes. Furthermore, the row vectors can be seen as a reception acknowledgment, because the vectors indicate the events that other processes have already received. In our example, row 1 indicates that process p from whose knowledge the matrix was derived knows that all processes have received event $(A, 1)$.

In order to maintain a matrix clock, processes must disseminate information about which events they have received. This information can be explicitly included when disseminating events or derived from event metadata.

2.2.4 Causal Order Delivery Protocols and their Applications

The causal partial order can be used to define a message delivery protocol. As an alternative to FIFO or total order delivery, **causal order delivery** delivers messages

in their causal order. This means that a particular message is only delivered after all of its (local and remote) causal predecessors have been received and delivered. Causally concurrent messages can be delivered in any order.

Causal order delivery has been used to build **communication middleware** that transparently delivers message in causal order. The practical merits of this use as a transparent reliable communication facility have been controversely discussed in a debate between Cheriton/Skeen and Birman [23, 13, 24] and others [26, 117].

Causal ordering has also been investigated in the context of **mobile systems**. This work was mainly concerned with causal ordering for infrastructure networks that consist of mobile systems and their support station infrastructure [5]. For this kind of network, Prakash et al. [91] describe an efficient causal ordering protocol.

Causal delivery per se only defines the order in which messages are delivered at their destination. The concept of causal delivery has been extended to include guarantees for the case of a crash-recovery failure model, where failed processes can recover and join the system without losing all their state [7]. **Causal message logging protocols** aim to capture causal relationships persistently [12, 8].

2.3 Data Replication

Replication is the act of copying data to multiple hosts and coordinating any changes in a way that makes the data copies appear to function like a single copy. Due to this inherent distributed nature, any replication system is confined in a system of **fundamental trade-offs** between consistency, availability and performance. Consequently, the perfect replication system (strong consistency, high availability and performance) can not be constructed [45]. Replication becomes a design problem that requires careful co-design of algorithms and system primitives. Algorithms and system primitives have to be adapted to the requirements of a particular application and their development is ultimately driven by systems requirement if the result shall be deployed as a production system.

The **replication algorithm** is the defining feature of a replication system. The algorithm's primary task is to maintain the consistency of the replicated data. This consistency is at stake every time a data replica is changed. When a single client accesses the replication system, any resulting inconsistency can be reconciled by reproducing changes at all replicas and ensuring that further reads access the latest data. However, if two or more clients intend to access the replicated data at the same time, the preservation of consistency becomes a non-trivial problem. In order to ensure consistency, the replication algorithm must control access to the local replica by its clients and orchestrate reads and writes with other replicas.

In the context of replication systems, the concept of **consistency** has been introduced to acknowledge the fact that in practice the physical representations of the replicated data on each of the replicas deviate much of the time. Replication systems guarantee a form of consistency of the replicated data that is defined from the perspective of its clients in a **consistency model**. The consistency model describes

invariants that the replication system maintains for the **interaction of local and remote reads and writes** to the replicated data. In this sense, consistency means the correctness of the replication algorithm with respect to a specification as given by the consistency model.

In addition to referring to the interactions between operations on the replicated data, the concept of consistency is also used in a second, slightly different meaning. In this second meaning, consistency refers to the **physical equality** of all the copies of the replicated data: two replicas are **consistent**, when their data replicas are equal. Depending on the consistency model, a replication system can be (temporarily) inconsistent with respect to the equality of the data replicas, but still operate correctly within the guarantees of the consistency model. However, in the absence of changes and given the communication necessary to reconcile any differences, all replication systems aim for the physical equality of replicas in order to guarantee the safety of their entrusted data.

The replication algorithm ensures consistency by joining into the interaction between the application and the data replica so that it can suspend the application's access temporarily and control what data it reads and writes. While exercising this control over the application, the replication algorithm can communicate with its peers and do what it is necessary to preserve the consistency of the data. The degree to which remote coordination and local control are coupled directly determines the characteristics of the replication system.

By design, **synchronous replication algorithms** couple concurrency coordination with consistency coordination: any access of the client to the replicated data is coordinated with other replicas before the result of its operation is returned to the client. Replicas control the progress of their clients and can thus define the interleaving of all accesses to the data. A synchronous coupling between clients and the replication system allows the system to give strong guarantees on the consistency of data. However, it also limits the size of the replication system to a handful of replicas and introduces a strong performance and failure dependency as it couples the progress of an individual replica to the progress of its peers (limiting availability in case of failures).

Asynchronous weak-consistency replication can eliminate many of the drawbacks of synchronous replication by decoupling local client access from the coordination of consistency. Instead of single-copy semantics, the design of a weak-consistency replication algorithm is determined by the semantic requirements of a particular application. The loose coupling between control and coordination gives weak-consistency replication its favorable properties: it is able to scale to large replication systems and usually has only a loose failure and performance dependency between its hosts.

2.4 Synchronous Replication

Synchronous replication algorithms can guarantee **sequential consistency** [85] through synchronous coordination of accesses between replicas. Sequential consistency provides clients with the illusion of working on a single copy of data instead of replicas. In its original context of multiprocessing systems, Lamport [76] defined sequential consistency as:

Definition 2.7 *A system provides* sequential consistency *if the result of any execution is the same as if the operations of all processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program.*

The two most important types of synchronous replication architectures are based a central sequencer or on quorums, respectively. When implemented for a realistic system model, they both require some form of fault-tolerant total order protocol [33]. The total order problem is similar to the distributed consensus problem and can be solved with algorithms such as Paxos [77, 78, 15].

Sequencer-based algorithms designate a replica that acts as the sequencer for all changes in the system. Also known as master-slave or ROWA (read-one-write-all) designs, these systems use leader election or lease negotiation algorithms [79, 65] to reliably determine the sequencer even in presence of failures. Alternatively, chain replication [118] resorts to restricting the communication topology to a chain that implicitly designates a sequencer.

Instead of relying on a designated sequencer, **quorum-based algorithms** use the decision of a majority to agree on a certain order of client accesses to the replicated data and thus synchronize accesses. Since the initial proposal of using quorums [44], numerous researchers have proposed quorum-based algorithms (such as [41]) or built systems that use quorum-based consensus decisions (such as [17, 20]). However, because each access to the replicated data needs to be negotiated and requires communication, research indicates [67, 51] that sequencer-based approaches outperform quorum-based algorithms in most cases.

Implementation considerations aside, synchronous replication algorithms must coordinate any change to the replicated data before accepting it. Through this coordination, they avoid any concurrent and potentially conflicting changes to the data. Hence, strongly consistent replication is also referred to as **pessimistic replication** because it prevents all potential conflicts.

2.5 Asynchronous Weak-consistency Replication

In order to eliminate some of the drawbacks of synchronous replication (strong coupling, limited scalability), weak-consistency replication foregoes the immediate coordination of each change. Due to this lack of precautious coordination, weak-consistency replication is also called **optimistic replication** because it optimisti-

cally assumes that coordination is mostly unnecessary and conflicting changes can be handled later in some way.

The lack of coordination in weak-consistency replication also implies that the replications system can only reach a consistent state when the changes eventually stabilize. Thus, the data replicas of weak-consistency replication system are usually not equal while they are modified. However they eventually become equal when all differences have been reconciled and no more changes are in flight. Because of this characteristic, weak-consistency replication systems are said to provide **eventual consistency**.

Note the contrast to strong-consistency replication systems, which are able to guarantee the immediate and continuous consistency of data replicas.

Weak-consistency replication algorithms generally fall into two categories, depending on whether their key element is the replicated data itself or changes to the same:

- **State-centric replication algorithms** directly compare the state of two replicas and infer the modifications that are necessary to lead them to mutual consistency.
- **Update-centric replication algorithms** track and record updates to the replicated data (usually in logs), disseminate these changes to other replicas and use them to update their data.

2.6 State-centric Replication

Instead of tracking and recording changes to replicas, state-centric replication algorithms compare the state of two replicas in order to infer the changes that are necessary for replica consistency. A state-centric replication approach has to solve the following problems:

1. *detect changes*: given a pair of data items, detect if the items have changed and how to update them. This includes the problem of distinguishing deletions from creations: when a data item is present on one node, but not on another, has it been created or deleted?
2. *infer necessary modifications*: given a mechanism for detecting changes for a pair of data items, how can we avoid comparing all pairs?

In order to detect modifications between data items, most state-centric approaches use some kind of summary that represents the causal history of each data item, such as version vectors or other representations (see Sec. 2.2.2). Version vector-based approaches to state-centric weak consistency [30, 69, 54] have been widely used for reconciling changes to optimistically replicated data.

The primary application domain for these approaches has been replicated file systems that allow disconnected operation, such as Ficus [111], Rumor [55] and

Microsoft's WinFS [81], where a version vector imposes minimal overhead when compared to the size of single files. Apart from file systems, the version vector-based approach has been also applied to the optimistic replication of databases [36]. The basic version vector protocols have also been used to synchronize devices via the industry-standard XML-based synchronization protocol SyncML [18].

These systems detect conflicting changes by comparing a version vector that is maintained for each data items [95], lock the resulting conflicting versions and make them available for manual (user input) or automatic (merge procedures) conflict handling. One of the focal points in this context is the handling of the various possible combinations of conflicting changes to a data item. The create-delete ambiguity is resolved using tombstones that represent deleted data items. Tombstones require additional means to garbage collect them after some time. In [30] and [69] Parker et al. also introduce a graph representation of the divergence of an optimistic replication system into mutually inconsistent partitions.

An alternative system for the optimistic replication of files is the Tra file synchronizer by Cox and Josephson [27]. The main focus of Tra is to solve the create/delete update problem without using tombstones. Instead of using a single version vector per data item, the authors propose to maintain a pair of version vectors per file that effectively capture the file's causal history of change operations (modification vector) and general events (communication vector), respectively. They observe that these vectors are intimately related because they represent the same causal history and can therefore be compressed. They also note that version vectors can be flattened if all files of a replica are synchronized at the same time. The Tra protocol has also been applied to synchronizing mobile devices with the industry-standard SyncML protocol [116].

In [29, 28, 27], the authors identify the problem of concurrency of conflict resolution and conflicting changes. While earlier work always assumed that a resolved conflict draws information from all of the conflicting versions, the authors posit that some conflicts are resolved by simply choosing one of the versions and thus explicitly ignoring concurrent changes. They interpret this resolution-by-copy as the permission to ignore further concurrent changes, saving the user the annoyance of further conflicts. Greenwald, et al. have also proposed alternative approach for communicating the result of conflict resolutions to all peers by sending so-called agree events [52].

In order to find out which parts of the data might have changed, the dependency-tracking metadata of two replicas has to be compared and transferred for that purpose. The full transfer of metadata can be avoided with the so-called set reconciliation protocol [84], which can avoid transfer of the full metadata. Breaton [16] describes an alternative approach to state-centric replication that prevents files from being changed except when the majority of replicas are online.

Because state-centric replication algorithms do not have any information about how a piece of data has changed, they need to transfer a full copy of the data in the naive approach. With partial checksums over the data [88] or by using erasure codes [66], changes can be inferred and the transfer volume can be reduced.

2.6.1 Discussion

State-centric replication algorithms have been mainly used for reconciling changes to replicated file systems. In this context, the file data will dominate any transfer and storage volume, and the efficiency of the replication algorithm does not play a huge role. This enables the use of per-file version vectors and protocols that require the transfer of the complete metadata during the reconciliation process.

In later sections we will present our algorithm for recreating version branches consistently on all replicas (Distributed Consistent Branching algorithm). From an application perspective, this algorithm provides a similar solution to the problem of handling conflicts as Ficus and Tra propose it in the context of file replication. Their notion of replica divergence is similar to our concept of branching. However, because our algorithm is update-centric, it has several advantages over previous approaches. In particular, it is more efficient in change detection as it does not have to transfer the complete metadata (such as Ficus and its successors) or traverse the hierarchy (Tra).

2.7 Update-centric Replication with Logs

An update-centric replication algorithm solves the replication problem by monitoring the local operations on each of the data replicas and recording information about them. While the specific way of this change record is different among replications system, it needs to contain enough information to reproduce the operation at a remote replica. Some systems capture changes on the database level as primitives that operate directly on data level ('set row to ...', 'add 5 to ...'), while others record application-level operations ('create calender entry'). Also change records do not necessarily need to contain the result operations (the target value) but may only be used to invalidate data items in order to allow their consistent update later by retrieving their latest state [10].

The replication system disseminates these changes records immediately or later in batches. When a replica receives remote updates, it applies them to its data replica while adhering to the given consistency guarantee.

The main data structure of an update-centric approach is a persistent log of changes. This log serves multiple purposes:

1. *It acts as the persistent and authoritative storage of changes* – all changes that have been made to the data are present in the log and all logged changes are eventually present in all replicas.

2. *It allows efficient lookups of differences* – given the state of a remote replica, it allows a process to efficiently retrieve all changes that are necessary to bring the local replica up-to-date.

2.7.1 Update Dissemination: Gossip, Epidemics and Rumor Mongering

Compared to directly sent messages, the use of logs allows more flexibility in how to use communication. Because messages convey updates instead of coordination information, protocols can include updates from other replicas as well as their own in transmissions to peers. The term **gossiping** has been adopted for this style of communication where hosts include information from other processes in their messages.

The concept of gossiping has been advanced to embrace randomized message exchange rather than simple deterministic ones. This **epidemic communication** has been shown to lead to convergence when applied to data replication [34] (rumor mongering), and has further been used for scalable communication protocols [38, 119] such as in reliable multicast [14].

Disseminating updates via gossip poses the immediate question of how to bound the growth of event histories. Wuu and Bernstein were among the first to describe such a protocol in the context of updating a replicated dictionary [121]. Their solutions requires all replicas to maintain a matrix of timestamps. The two-phase gossip protocol [58] disseminates two timestamp vectors to determine which events need to be disseminated, which events can be deleted from the local history, and which events can be applied to a checkpoint. [93] claims to optimize the update dissemination process further so that the gossip protocol runs only for a complete database replica instead for single data items.

Agrawal et al. [3, 60] apply log-based gossip communication to the replication problem for databases that need to provide transactional integrity and serializability. Manassiev and Amza introduce dynamic multiversioning [82] to make read-only transactions scalable.

Golding uses log-based gossip-style update dissemination to build a weakly consistent group membership protocol with reliable, eventual delivery called the TSAE (time-stamped anti-entropy) protocol [48, 47]. He applies this protocol to the problem of building a large-scale replicated information system for BibTeX records, refdbms [49].

2.7.2 Ordering of Updates

In order to achieve consistency, an update-centric replication system has to apply local and remote operations at each of the data replicas. The order in which the operations are applied determines the effective semantics of the operations. Hence the rules by which the operations are ordered implicitly define the consistency model of an update-centric replication algorithm.

In this context, the task of an update-centric replication algorithm is to determine and enforce an order or **schedule** of operations that is consistent across replicas. The rules by which this schedule is determined have to account for the application's requirements for the semantics of changes. There are two basic ways of determining

a schedule for operations, semantic and syntactic ordering [101]:

- **Syntactic ordering** mechanisms order changes solely by when, where and by whom operations have been submitted. They have no further insight into the changes made or the data they affect and consequently operation semantics are not considered when determining a schedule.

- **Semantic ordering** mechanisms explicitly depend on information from the operations and rely on introspecting the semantics of the replicated data and the changes to it. They are able to exploit semantic properties such as commutativity and idempotency of operations and reorder or suppress operations based on their semantical meaning to the application in order to avoid conflicts or roll-backs.

In order to be able to reorder operations and thus generate different schedules, semantic replication algorithms have to record the **operations** proper instead of merely capturing changes to the data. The recorded operations can later be applied to remote replicas to transform their data. In contrast, syntactic ordering mechanisms only capture the effect of operations as **changes** and disseminate them. Because the differentiation between the terms update, change and operation is irrelevant for this taxonomy, we will use the terms interchangeably.

Syntactic Ordering Algorithms and Systems

Lazy replication [73] comprises a set of gossip-based protocols that allow applications to choose from a set of predefined consistency guarantees. To implement these guarantees, updates are given unique identifiers and clients have to explicitly specify which updates a read or query operation has to encompass. Using these identifiers, normal operations are causally ordered. The system further supports so-called forced and immediate updates, which implement total (causal[2]) ordering between changes, but still use the same read and query operation.

The work on lazy replication lays the foundation for the Eventually-Serializable Data Services (ESDS) [40]. Clients of ESDS interact asynchronously with the replication system and specify whether any issued operation needs to be a stable part of a total order at the time of acknowledgment or can be later re-ordered. Beyond this choice, the client is not further exposed to the internals of the replication system. Cheiner et al. [22] describes a mapping of the formal description of ESDS in [40] to an implementation and analyze the performance of an implementation that has resulted from this work.

The OSCAR replication system [35] defines an ordering protocol that can be configured for operations that are commutative and associative, overwrite operations

[2]We stress causality here because it allows us to give sequential consistency guarantees. Non-causal total order destroys data dependencies between operations. While non-causal total order can be advantageous for some application domains [104], it does not translate directly to familiar consistency guarantees as causal total order does.

with an empty read set and site-sequential operations that are totally ordered when they stem from the same source.

The Bayou project advocates a non-transparent approach to replication that involves the application in the replication process [114]. To that end, Bayou introduces a transaction mechanism that lets applications express data semantics in an implicit way [113]. Instead of issuing simple updates to the data, the application adds a precondition that names the state of data it assumes the change will be applied to (see Fig. 2.5). This precondition is expressed as a query along with its expected result and is part of each change operation. The application also supplies a merge procedure with each update that specifies the system behavior when the precondition could not be met.

```
Bayou_Write(
  update = {insert, Meetings, 12/18/95,
            1:30pm, 60min, ‘‘Budget Meeting’’},

  dependency_check = {
    query = ‘‘SELECT key FROM Meetings WHERE day = 12/18/95
              AND start < 2:30pm AND end > 1:30pm’’,
    expected_result = EMPTY},

  mergeproc = {
    alternates = {{12/18/95, 3:00pm}, {12/19/95, 9:30am}};
    newupdate = {};
    FOREACH a IN alternates {
      # check if there would be a conflict
      IF (NOT EMPTY (
      SELECT key FROM Meetings WHERE day = a.date
      AND start < a.time + 60min AND end > a.time))
      CONTINUE;
      # no conflict, can schedule meeting at that time
      newupdate = {insert, Meetings, a.date, a.time,
                   60min, ‘‘Budget Meeting’’};
      BREAK;
    }

    IF (newupdate = {}) # no alternate is acceptable
      newupdate = {insert, ErrorLog, 12/18/95, 1:30pm,
                   60min, ‘‘Budget Meeting’’};
    RETURN newupdate;
  }
)
```

Figure 2.5: A write operation in Bayou as it is issued by an application and disseminated by the system (taken from [113]).

Bayou orders changes with the help of a primary commit scheme and exposes two views on the replicated data that represent the result of tentative and stable operations. The system accepts and applies all changes to the tentative state immediately in the order they are received. On change reception, a designated primary node defines a total order on operations that defines the stable state and disseminates it. In [90] Petersen et al. describe how the augmented transactions are disseminated with a protocol that is similar to two-phase gossip and TSAE. While Bayou itself is

targeted at mobile Computer Supported Cooperative Work (CSCW) applications, Bayou's augmented updates were later successfully applied to WAN replication in the context of the Oceanstore [72] project.

Early work in the Bayou project resulted in *session guarantees* [112] that provide clients of a replication system with ordering guarantees for both client reads and writes. These guarantees are ensured in interactions across multiple replicas by maintaining a stateful relationship with the replicas of the replication system. Applications can specify that reads need to reflect all previous writes or reflect a non-decreasing set of writes. Writes can be ordered in a way that causality between write operations is maintained and that writes always follow their reads. All guarantees are implemented by assigning a unique write identifier (WID) to changes. These WIDs are used to indicate the current read and write set of the application, which is summarized in a version vector as part of the session state. In their design, session guarantees bear some similarity to the concept of snapshot isolation from the database literature [59].

The Hierarchical Asynchronous Replication Protocol (HARP) [2] uses gossip communication in a tree topology to achieve scalability. The focus of the work is how to maintain this tree structure under churn. HARP supports several delivery methods for operations: unordered, latest-wins, node-local FIFO order and total order.

Baldoni et al. [9] propose a protocol that minimizes the number of rollbacks necessary for establishing a total order among operations. The protocol has only been evaluated in a simulator, with a 1000-node replication system and a peak broadcast rate of about 25 broadcasts per cycle.

Saito et al. designed and implemented Porcupine, a clustered and replicated mail server [102]. Its weak-consistency replication mechanism [100] transparently manages data objects (syntactic approach). Objects are modified with an update mechanism that also specifies the replica set of the object and permits object deletions by supplying an empty replica set. The replication algorithm differs from other algorithms in that it does not use a log, but sends change events directly to all peers. Concurrent changes are arbitrated by choosing the one that has been issued later according to wall clock time.

Semantic Ordering Algorithms and Systems

IceCube [71] introduced a generic scheduling mechanism that orders changes according to application-defined constraints. Consequently, the ordering problem is modeled as constraint satisfaction problem whose constraints are supplied by the application. Compared to syntactic approaches, this approach has a great deal of freedom in ordering operations and can thus avoid many potential conflicts. Because operations constraints are not specified as part of an operation, IceCube has a more lightweight format for logged changes than Bayou. The idea of using constraint programming for devising a schedule for operations was further refined in several directions: in Hamadi and Shapiro [56], the authors investigate efficient solutions

to the computationally intensive problem of computing a schedule that satisfies the constraints. In other work [108], constraint-based scheduling has also been applied to reconciling changes in a mobile file system.

Gao et al. [42] present a replication protocol that is able to respect application-defined operation semantics and show how to scale a replication system for the TPC-W benchmark, which implements an online bookstore such as Amazon's. For each data type in TPC-W, such as customers and orders, Gao defines custom consistency requirements. An implementation has been thoroughly benchmarked with a TPC-W workload.

Read Consistency

While syntactic and semantic ordering is mostly concerned with the ordering of writes, Bayou's session guarantees have already introduced a notion of consistency for reads. This aspect of consistency has been further refined by work on the TACT middleware and on PRACTI replication.

The TACT middleware by Haifeng and Vahdat [123, 122, 124] enforces consistency on the level of consistency units (conits, which could be for example an airplane seat in a reservation system). TACT is able to bound and control the consistency error for a conit. The consistency error can be specified as a tuple (numerical error, order error, staleness), where numerical error describes a numerical difference to the value as strong consistency would have achieved, order error captures the number of writes that might require reordering and staleness describes the time a replica may lag behind current state of other replicas.

Bayou-based PRACTI replication implements a protocol that can provide topology independence, partial replication and arbitrary consistency guarantees. The latter is achieved in a scalable manner by disseminating imprecise invalidations independent from the updates to the data. The application can exercise control over consistency by specifying how precise a read from the data has to be, i.e. how many imprecise invalidations the data may contain [31].

2.7.3 Global State and Log Maintenance

The eventual consistency promise of weak-consistency replication brings up the immediate question on when this eventual consistency has been achieved. The answer to this question is important in at least two contexts:

- *Change stability.* Some weak-consistency replication systems (like Bayou) can re-order changes as concurrent changes come in, and thus influence the outcome of the change. An application might want to know when its change is **stable** and will not be further touched by the system.

- *Garbage collection.* Apart from this external notion of write stability, the replication system itself is interested when an update can be considered stable. Ensuring consistency requires internal state that needs to be garbage collected

when it is no longer required. A prominent example for such an internal data structure is the communication log, where the associated problem is **log pruning** or cutting.

The stability of a change is a global property of the replication system. Its observation by each of the processes can be represented as a matrix clock (see Sec. 2.2.3). A naive implementation of matrix clocks for a system of n processes adds an overhead of $O(n^2)$ to each event [99]. Ruget [99] uses the concept of an antecedence graph [37] that captures the causal structure of the distributed system to maintain matrix clocks efficiently with a communication overhead of $O(n)$. He also introduces approximate representations of a matrix clock. For a hierarchically structured replication system, [68] presents a protocol that achieves an overhead of $O(\sqrt{n})$. Golding gives a protocol for an approximate matrix clock [48] that attaches a timestamp vector to each event (thus also $O(n)$).

Sarin and Lynch [103] discovered the garbage collection problem early and used a distributed snapshot algorithm to agree on which events are obsolete. Chittajullu and McMillin [25] describe the garbage collection problem and a solution for event histories in a monitoring system.

2.7.4 Logs as a Systems Data Structure

Update-centric replication relies on a log for ensuring remote and local consistency of data. Logs are known to be an efficient persistent data structure for data which is mainly written [120] and are obviously simple to implement through their append-only mutability. These observations have inspired researchers to investigate how logs can be applied to persistence problems in several domains. As the log is the core data structure of these systems, their architecture is said to be **log-structured**.

Rosenblum and Ousterhout proposed to use a log as the base storage structure for a file system [98]. This log-structured file system allocates disk blocks in a round-robin manner over the whole disk. Log-structuring has also been proposed for database design [80]. The Vagabond database system [89] uses a log-structured approach to the design of a temporal database, which allows users to retrieve old versions of data.

2.7.5 Discussion

Update-centric capture and dissemination of changes to replicated data has been applied to a wide range of replication systems. It allows the system to **only transfer actual changes** to the replicated data, which makes the protocol efficient. Furthermore, it is well understood that the underlying gossip-based communication allows eventual propagation to all nodes in the system and has mechanisms to determine how far updates have been disseminated.

Syntactic approaches use update-centric replication to order changes to replicated data in a static way that is independent from the actual semantics of the updates. Its replication mechanism can be presented in a simple programming model.

Semantic approaches give applications more influence on operation ordering. Their programming interfaces are specific to certain application domains.

Most research in this area has concentrated on the distributed algorithm side of the replication problem and kept system design and implementation aside. Consequently, hardly any performance measurements of real systems are available. The implementation of ESDS has shown that the transformation of a replication algorithm to a full implementation can be a complex task [22]. Generally, this transformation a challenging and while no general model has established itself yet it is an area of active research [11].

The ordering **algorithms in this thesis** (see chapter 6) follow a syntactic approach and record changes (instead of operations), but allow applications to influence their behavior. We do not attempt to provide a total ordering of operations. While total order allows weak-consistency replication to come up to par with strong-consistency systems, it can not be provided transparently in a non-blocking manner[3]. Instead we restrict ourselves to causal order (and thus gain availability of data) and show that it can be maintained efficiently both from an algorithmic and systems perspective. In order to make it useful in practice, we provide mechanisms for ordering concurrent operations while maintaining causal order.

In this dissertation, we propose to not only use the log as an abstract data structure for replaying updates, but also as the central persistence mechanism. **Log-structured storage** has already been successfully applied to database and file systems, is well-understood and is a good match to the physical constraints of storage media such as hard disks and flash. We will investigate the requirements of replication on the persistent log. In particular, we will present a mechanism for implementing operation ordering for an append-only log, show how obsolete information in the log can be identified (see also [62]) and how data in the log and the local database can be kept consistent ("local consistency").

[3]Weak-consistency systems that provide total causal order have to continuously reorder operations and require mechanisms to detect when a stable order is reached and access to the data can granted. This control over read access to the data reduces the availability of the data in order to guarantee stronger consistency.

Chapter 3

System Model and Assumptions

In this dissertation, we propose an approach to weak-consistency replication that emphasizes the system architecture aspects of the replication problem. As a consequence of this focus, we have to co-design

1. the distributed algorithms for replication,
2. the programming interface that controls the replication system,
3. and the use of the interfaces of the operating system that hosts our replication system.

In our *system model*, which is presented in this chapter, we capture the assumptions that underlie our architecture and algorithms and make them explicit. These assumptions are mainly in two areas:

1. The *abstracted environment* in which our distributed algorithms run, which is characterized by assumptions about the network and the hosts.
2. The *architecture of the application system* that employs our algorithms.

3.1 The System Model for Distributed Algorithms

The system model of the distributed system captures the abstract properties of the underlying physical host and networks that are relevant for the presentation and correctness arguments of the distributed algorithms that we will present later.

As a base model we consider a system of N nodes that consist of the application and an embedded (in-process) database. Nodes participate in distributed algorithms and protocols as *processes* that can be uniquely identified. Because we are designing a distributed system that maintains data, we assume that nodes have a persistent storage that can store a considerable volume of data. That implies that failed nodes that rejoin the system will want to update their local data replica instead of starting from a blank sheet, resulting in a crash-recovery model.

We assume a networking environment that demands weak-consistency replication techniques because of its long communication latencies and its possibility for failures up to network partitions, namely either servers connected to a wide area network (WAN) or mobile devices that communicate directly over wireless ad hoc network links. Both kinds of networks transmit data as *datagrams* of a bounded size. These datagrams can be duplicated and lost, but their content is protected against modifications. WANs and ad hoc networks might seem to be vastly different at a first sight, but they actually share key properties such as a long communication delays and a non-negligible probability of link and transmission failures.

Server in a Wide-Area Network

Wide area networks like the Internet are characterized by high bandwidths and potentially high communication latencies. For instance, an optical Internet link from Europe to the U.S. might be able carry many gigabits a second, but has a latency of at least 100 ms.

The connection between hosts in WANs are usually not direct physical connections, but are mediated by routers of the network infrastructure over multiple links. This infrastructure is usually able to tolerate failures of links or network equipment by routing around problems using redundant connections. In practice that means that datagrams can simply be lost or received in a different order than they were sent because they might have traveled over different paths through the network. There is also no guaranteed maximum time for delivering a datagram.

From the perspective of a distributed algorithm that means that algorithm can not be sure when or even if a datagram will be delivered to its destination. The algorithm has to make a more or less arbitrary assumption about the maximum round trip time of its datagrams and their responses. Hence, the algorithm can not safely distinguish failed hosts from hosts whose links have unusually high latencies. Furthermore, when multiple hosts are connected over a WAN infrastructure and parts of the system fail, the system can be split into multiple partitions, each of which is able to communicate; in this case the algorithm cannot decide with any certainty whether the unreachable hosts have failed or are simply located behind a bad link.

Mobile Wireless Systems

While wireless ad hoc networks can have latencies similar to those of WANs, they usually have less bandwidth and are less reliable. Due to the mobility of nodes and the inherent unreliability of wireless links, the set of nodes that can be reached from a certain node is highly dynamic, and the communication link between two nodes may be asymmetric and can disappear at any time. These assumptions hold true for a wide range of wireless network technologies, from wireless networks with infrastructure (like GSM, 802.11) to wireless ad hoc networks (802.11, Bluetooth) to personal area networks (PANs).

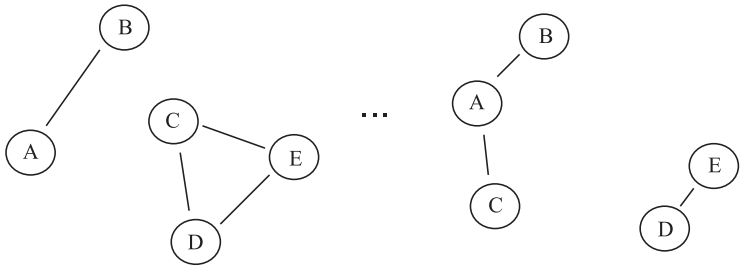


Figure 3.1: The communication graph of a mobile system changes over time and can be partitioned

While in WANs network partitions are a rare case (which still must be considered), it is the common situation for processes in an ad hoc wireless infrastructure. Thus the communication graph that captures the mutual reachability of mobile devices may be sparse and can change quickly over time (see Fig. 3.1).

Resulting Model

To summarize, we assume the following properties of a distributed system in the design of our distributed algorithms:

- Messages can be lost or duplicated, but message content is not altered (checksums).
- We assume a crash-recovery failure model, assuming processes with persistent storage.
- We operate as an asynchronous system with no timing considerations.

Considerations of lack of trust, including byzantine behavior are not in the scope of this thesis and were therefore omitted.

3.2 Application Model

Our replication algorithms facilitate the implementation of replication systems for applications that have certain characteristics. These characteristics are mainly related to the general application architecture and the data and programming model that we can support.

3.2.1 Application Architecture: Synchronous Serial Data Access

Our first assumption is that application instances are co-located with their data replica and access it in a synchronous, strictly sequential manner. In practice, this kind of application architecture can be found in applications that use an **embedded**

database engine in each application instance or rely on dedicated database instances as their storage layer.

In case the application has internal concurrency, we assume that it arbitrates any access to its local data replica in a way that results in a strictly sequential access, usually via **mutual exclusion** of threads. In effect, we do not investigate concurrency control within a replica (for which a huge body of work from the database community exists) and across replicas.

The latter implies that we also assume that the replication system is **not used as the primary coordination method** between application instances, and the application relies instead on internal means to coordinate concurrency. While our augmented causal consistency model arbitrates concurrent accesses, applications that need to primarily rely on the replication system for that purpose are better served by other replication approaches.

We have to make these assumptions for several **reasons**. The assumption of synchronous serial access to each data replica is due to the fact that causal consistency inherently assumes single principals that change the data and therefore we concurrency within the replica can not be supported. Our assumption of operating in an unreliable networking environment makes it necessary that applications are co-located with their data replica so that the data replica is always accessible. Similarly, an unreliable network also hinders the communication between replicas and so transparent coordination of application instances over the replication system is not feasible.

In practice, our assumptions are satisfied for **mobile systems** that run Personal Information Management (PIM) or Computer-Supported Collaborative Work (CSCW) software. In these systems, the data access is usually driven by the interactions of a human operator and have at most have a low level of concurrency. This directly translates to synchronous serial data access. Furthermore, the lack of a reliable network dictates a server-less environment with local data storage.

Our algorithms are also targeted at **server applications** that interact with the database system in transactions that are purely functional way, i.e. their transactions do not interact with the external environment. Consequently, this restricted class of transactions can be collapsed to a simple *read from the database – compute – write to the database* pattern. As there is no synchronous interaction of the transaction with the environment, the execution of the transaction can be fully offloaded from the application to the database engine, and it can be processed atomically and in an isolated manner without blocking. Bayou's transactions (see Fig. 2.5) are an example of this style of interaction with the database.

As an example of an application that matches our profile, consider a large-scale web service such as social networking or web mail, in which users interact mainly with their profile in a session, and where cross-session interaction (such as writing messages) is either excluded by system architecture via some way of mutual exclusion or handled by application protocols. In section 6.4.2, we discuss classes of applications that fit to our assumptions in more detail.

The assumptions that we make about the system architecture most notably **ex-**

clude applications that require data access with general ACID-style (atomic, consistent, isolated and durable) transaction guarantees as they are provided by classic RDBMSes. ACID transactions provide a very general model of computation [46] that not only includes computing a function from its input, but also allows interaction of the computation with the outside world. For this fine-grained synchronization and coordination between application instances, the RDBMS provides the means in a centralized or cluster-replicated architecture and therefore strongly relies on reliable, low-latency communication[1].

3.2.2 Programming and Data Model

Dictionary data. Generally, our algorithms expect to work on data which can be structured as a dictionary (unique keys that are associated with data values). This data structure should allow the implementation of a wider range of data models, from simple record databases over trees and hierarchical structures to relational data models

Causal consistency. In case there is no concurrency in the causal sense, we guarantee causal consistency for access to the data. This means that in this case, any data dependencies between operations are preserved and operations behave as if there was only a single copy of the data. We extend the causal consistency guarantees to handle concurrent access by introducing a branching semantics (for details refer to chapter 6).

Opaque data semantics. Our replication algorithms treat the data that is associated with each key in a transparent manner and do not require it to have a certain structure. They also do not rely on any semantic information from the data.

While this approach inhibits use cases that would profit from application semantics, it is able to present a much simpler programming model to external applications. Consequently, the programming interface looks like that of non-replicated record databases with simple read and write accesses to keys and applications that have been designed for key-value interfaces can be ported to use a database that is based on our framework without major effort.

[1] See also [110] for a discussion of application domains of classical RDBMSs.

Chapter 4

Tracking and Disseminating Changes

4.1 Overview

Over the course of the next few chapters we propose an approach to causal replica consistency that models changes to replicated data as events in a distributed system and preserves data dependencies between changes as causal dependencies between the events. Our approach is based on a persistent, shared log of local and remote changes.

In this chapter, we begin our presentation by describing the representation of changes of the replicated data as events in a distributed system (Sec. 4.2). We show how the causal dependencies between events can be captured in dependency vectors, and how these dependency vectors can be used in developing a notion of a local view on the global state of the replication system (Sec. 4.3).

The representation of changes as events is the foundation for two protocols that track changes and disseminate change events to remote replicas in a structured manner, **causal gossip** (Sec. 4.5) and **direct send** (Sec. 4.6). Causal gossip is a gossip-based mechanism for offline replication in mobile systems, while direct send is an online algorithm that broadcasts changes directly to its peers. Both protocols preserve data dependencies between changes and allow systems to ensure the consistency between replicas as long as no concurrent changes are made. Furthermore, they provide the primitives and guarantees that our consistency algorithms in the next chapter can rely on.

4.2 Representing Changes to Replicas

In the following we will show how we model distributed changes to replicated data. The presentation directly connects to the terms and concepts that have been introduced in section 2.1. However, as we are modeling a replication system, we apply the process concepts on the level of changes to one data replica and their dissemination to other replicas.

4.2.1 Changes and Data Dependencies

We have defined our target replication system as one that co-locates the replicated data and the application that accesses the data on the same host. Furthermore, we have specified that the application modifies the replicated data sequentially. In such a system the application's access to data follows this process (see Fig. 4.1):

1. the application reads data from the local replica,
2. the application performs its operation and modifies the data,
3. the application writes the modified data back to the local replica.

On the local replica, this sequence of application operations results in a change of the local data. In effect, the replica performs a state transition from state N to state $N+1$. The application is connected to this state transition with data dependencies. The read of the application creates a read dependency to state N, writing back the modified data creates a write dependency to state $N+1$. In effect, the application has changed the state of the replicated data synchronously.

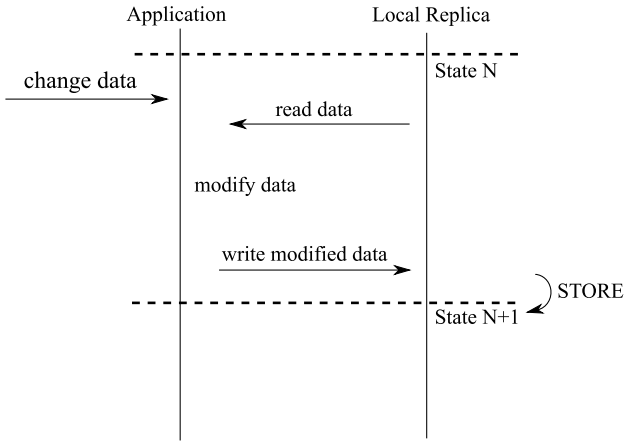


Figure 4.1: Synchronous coupling of application and replica.

From the perspective of the replication system, the new state $N+1$ is directly causally dependent on the state N. This state transition is a local event in the sense of the definitions of 2.1 and can be identified as such. The replication system captures this modification for later dissemination and recovery purposes by logging the changed data as a **change event** along with its causal dependencies. This change event is a complete description of the change that has been made. Its causal dependencies record the context in which the change has occurred and in which it can be safely applied to a remote replica. Only changes that were made concurrently need detection and special handling.

The change event does not contain the application operation proper, but only captures the operation's effect on the local replica. While it is possible to build

algorithms on top of changes that contain application operations (which for example IceCube [71] does and to some extent Bayou [113]), we chose to only capture the changes.

Capturing the changes has the advantage that the latest state of the data is entirely defined by the latest operation and does not have to be computed by executing all its data-dependent predecessors. Thus we do not need causal delivery in order to be able to correctly reproduce the changes at remote replicas. Because the latest change subsumes all previous changes, changes do not have to be delayed until causal delivery is ready. Instead each change can be delivered immediately.

Using this definition, operations are equivalent to events with one exception. Because operations are used to communicate the change of a replica to its peers, we can bundle multiple changes to **batches of operations** and treat them as one event if there has been no communication with other replicas in the meantime. When batching operations into one event, we have to maintain their order because the data dependencies within the batch are valid and important.

In the sections that follow an operation o is sometimes used interchangeably with the event e that transports it. Because of this containment, most of our definitions for events can be applied directly to operations and implicitly refer to the event that frames them. For example, $id(o)$ names the event identifier $id(e)$ of the event e that contains o.

4.2.2 Representation of Causality as Dependency Vectors

The order of subsequent changes to the replicated data have to be preserved in order to be able to reproduce the captured changes correctly at all replicas. We capture these data dependencies between changes as causal relations between change events. In order to make these causal dependencies between change events accessible to algorithms, we map them to a vector-based representation of causality.

As it operates, a process receives change events from its peers that influence any further operation. Let $latest_p(q)$ be the event identifier of the latest event that process p has received from process q. Per definition, the causal dependencies of an event relate to what the generating process knows when it generated the event, i.e. which events it has received and processed before. This knowledge can be summarized in the *summary vector* of the process, which keeps the event identifier of the latest events, $latest_p(q)$.

Definition 4.1 *The* summary vector $S_p(t)$ *of a process* p *at logical time* t *is a function* $S_p : P \to \mathbb{N}$ *that assigns each process* $q \in P$ *the timestamp of the latest event that* p *has received from the particular process:*

$$S_p(t) := q \in P \to time(latest_p(q)).$$

As the summary vector of the process names the direct causal predecessors of an event, we can use it to capture an event's causal context. To that end, we associate a *dependency vector* with each event that represents the summary vector at the event's genesis.

Definition 4.2 *The* dependency vector $dep(e)$ *of an event* e *from process* p *is the summary vector of the process at its generation time:*

$dep(e) := S_p(t)$, *for* $p = process(e)$ *in state* $time(e)$.

The dependency vector only names the direct causal predecessors of an event, and does not allow us to draw conclusions about the causal relationship between two events without further guarantees. We will describe how to interpret the dependency vector in a later section, in the course of presenting our dissemination protocols.

Example Assume the latest events received by a process p at time t were:

$$e_A \text{ with } id(e_A) = (A,5) \text{ and } dep(e_A) = \begin{pmatrix} 5 \\ 2 \\ 1 \end{pmatrix},$$

$$e_B \text{ with } id(e_B) = (B,9) \text{ and } dep(e_B) = \begin{pmatrix} 3 \\ 9 \\ 2 \end{pmatrix},$$

$$e_C \text{ with } id(e_C) = (C,7) \text{ and } dep(e_C) = \begin{pmatrix} 8 \\ 3 \\ 7 \end{pmatrix}.$$

Then its summary vector becomes

$$S_p(t) = \begin{pmatrix} 5 \\ 9 \\ 7 \end{pmatrix},$$

which is also the dependency vector $dep(e)$ for a change event created in the transition from t.

4.3 Global Knowledge

Our algorithms use the shared log of events as an append-only data structure. In a naive implementation of these algorithms the log would grow without bounds if we would not free it from obsolete events from time to time. Identifying and removing obsolete events from the log is the task of **log pruning**.

Log pruning is dependent on a notion of the obsolescence of events that suits all algorithms that rely on information from the log. Fortunately, all of our algorithms can rely on the same concept for declaring an event obsolete: which processes know the event for how long.

The knowledge of processes about events can be captured with a timestamp matrix (ref. Sec. 2.2.3). We will now show how we can derive a timestamp matrix from causality information as we receive it. Using matrices derived from this

information, we will define two levels of global knowledge, first and second order acknowledgments. These two levels of acknowledgments will serve as a criterion for our algorithms for distinguishing required from obsolete information in the persistent log.

4.3.1 Deriving Acknowledgments from Causal Information

A causal dependency of an event implies that certain events were known by a process when the event was created. According to the definition of the dependency vector, it names all the event's direct causal predecessors. Consequently, we can interpret the dependency vector of received events as an reception acknowledgment for at least all events that are named by the dependency vector.

The reception of the change event $latest_p(q)$ from process q by process p conveys the information that process q has received at least all of the events named by the dependency vector $dep(latest_p(q))$. Therefore the change event $latest_p(q)$ *acknowledges* the reception of all events that are included in $dep(latest_p(q))$ by the process q.

When the system's dissemination protocols have delivery guarantees, the delivery of an event can imply that other events must have already been delivered and we can transitively infer that other events must be know by the node, too. In a system with a causal delivery mechanism, an entry in an event's dependency vector implies that all causal predecessors must have been delivered already and therefore all events with an identifier contained in the dependency vector were known by the process when the event was created. Similarly, with reliable FIFO delivery all event from one process are delivered in order and without gaps. Then the dependency vector also implies that all events contained in it have been received.

The dependency vectors of the latest received events for all nodes can be combined as column vectors of an *acknowledgment matrix* A.

Definition 4.3 *The* acknowledgment matrix $A(p)$ *of a process p is composed by the column vectors $dep(e)$, $e = latest_p(q)$ for all known processes q.*

The latest received events from any node therefore contain the latest acknowledgments for what the particular node has received. Combining this information allows a particular node to derive for any event, whether all participating nodes have already received that event. We formalize this as the i-know-that-all-know (or short: all-know) condition that can be checked for every event.

Formally, the all-know condition can be calculated as follows: Given an event e with the identifier $id(e) = (q,t)$, row q of the acknowledgment matrix A acknowledges the reception of the event by all nodes if $min_j(A_{qj}) \geq t$.

Definition 4.4 *The all-know vector of the node p, $a^1(p)$ is the component-wise minimum of $A(p)$'s column vectors $a^1_i(p) := min_j(A_{ij}(p))$.*

The all-know condition consists of a timestamp for every known node and therefor defines a cut through the history of events. Events before this *all-know cut* are known

to be known by all nodes, events after the cut might not have been received by every replica. We can formalize this as:

Theorem 4.1 *A node p knows that an event e with $id(e) = (q,t)$ has been received by all nodes, if $a^1_q(p) \geq t$. We call this a first-order acknowledgment.*

The acknowledgment matrix changes with every event that a process receives, as does the all-know vector a^1. In order to efficiently discern which events in the log satisfy the all-know condition, we can compute the log position that distinguishes events: it is the smallest log position of an event named in the all-know vector a^1. Events before this lower bound are causal predecessors to at least one of the events named by a^1 and therefore acknowledged by all nodes. This lower bound can be updated as new acknowledgments are received.

Example Assume the latest events received by a process were:

$$dep(e_A) = \begin{pmatrix} 5 \\ 2 \\ 1 \end{pmatrix} \text{ from process } A,$$

$$dep(e_B) = \begin{pmatrix} 3 \\ 9 \\ 2 \end{pmatrix} \text{ from process } B,$$

$$dep(e_C) = \begin{pmatrix} 8 \\ 3 \\ 7 \end{pmatrix} \text{ from process } C.$$

These dependency vectors can be gathered to the acknowledgment matrix

$$A(p) = \begin{pmatrix} 5 & 3 & 8 \\ 2 & 9 & 3 \\ 1 & 2 & 7 \end{pmatrix}.$$

The resulting all-know vector becomes

$$a^1(p) = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix},$$

which means that all processes know at least events $(A,3)$, $(B,2)$ and $(C,1)$.

4.3.2 Second Order Acknowledgments

The process of calculating the all-know vector can be generalized for computing a higher order of global knowledge. Using essentially the same process, we can

define the i-know-that-all-know-that-all-know condition, a second order acknowledgment. For an event that meets this condition, it implies that all nodes have received a particular event, and further that all nodes know that everybody in the system has received that event.

The definition of the second order acknowledgment derives from the first order one. Instead of using the dependency vectors from the latest event that we received, we use the dependency vectors from the events named by a^1. As a result, we get the second-order acknowledgment matrix A^2:

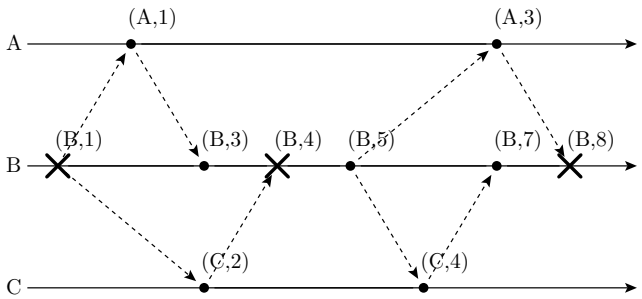
Definition 4.5 *The acknowledgment matrix $A^2(p)$ of a process p is composed by the column vectors $dep(e)$ for all e named by $a^1(p)$, i.e. $id(e) = (i, a_i^1(p))$ for all rows i of $a^1(p)$.*

From the acknowledgment matrix A^2, we can derive the second-order vector a^2.

Definition 4.6 *The all-know-that-all-know vector a^2 is the component-wise minimum of A^2's column vectors $a_i^2 := min_j(A_{ij}^2)$.*

We will use both first- and second-order acknowledgments later in our algorithms to determine different aspects of obsolescence in order to prune the log. For applications second-order acknowledgments can contain important information about the stability of changes.

Example Consider the causal dependencies for events from the process B that are implied in the following process-time diagram:



The event $(B,1)$ has been received by all processes at $(1,1,2)$. Acknowledgments for this fact reaches B at $(B,4)$. At this point the all-know condition a^1 for the event $(B,1)$ is satisfied (we can not make statements about the all-know condition for events from other processes at this point, because we have left out the causal information from the diagram for better clarity.). The event $(B,4)$ in turn implicitly conveys the all-know condition for $(B,1)$ in its dependency vector. When acknowledgments for $(B,4)$ have reached B in $(B,8)$, the a^2 condition for $(B,1)$ is met from the perspective of B.

4.4 Change Dissemination and Logging – Requirements

Events that carry the changes made at a replica have to be disseminated to other nodes of the replication system so that these systems can update their local data replicas. We present two event dissemination mechanisms, one for mobile replicas that may be offline for extended periods of time and one for online replication between servers.

Event dissemination must ensure that change events are accompanied with enough information for the receiver to compute data dependencies. Furthermore, the dissemination protocols have to deliver change events in an order that enables the receiver to apply the conveyed changes without violating data dependencies. Causal delivery provides these guarantees: it ensures that change events always arrive after their causal predecessors and thus overwrite their predecessor's change.

While causal order delivery is able to preserve change ordering, it can imply prohibitive delays for message delivery for online replication systems. Online replication systems usually disseminate messages with a broadcast protocol. A causal broadcast protocol has to defer message delivery until all of a message's causal predecessors have been delivered.

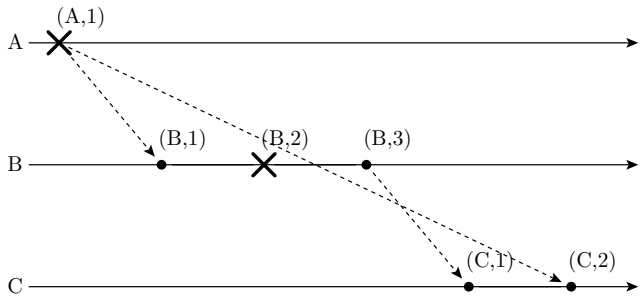


Figure 4.2: An event overtaking its causal successor

Consider the scenario in Fig. 4.2. Event $(A,1)$ is created and disseminated to processes B and C. B receives the event and creates and disseminates $(B,2)$ that reaches C before $(B,2)$'s causal predecessor $(A,1)$ has reached it. This can happen for example when the message to C was lost and (after a time out) retransmitted. When C receives $(B,2)$, its causal delivery mechanism would have to defer its delivery until $(A,1)$ has been received and delivered at $(C,2)$.

Thus in a causal broadcast, the message's travel time is dependent on the worst delivery time between all hosts instead of the normal message delay between two hosts. Furthermore, travel time can become dependent on temporary performance problems of any of the replica hosts. The result is a strong coupling of performance. For these reasons we do not require our dissemination protocol for online replication to provide causal delivery guarantees, but use a reliable FIFO protocol instead. Fig.

4.3 shows an overview of the properties of the two protocols.

protocol	causal gossip	direct send
domain	offline mobile replication	online server replication
communication	pull	push
	gossip/reconciliation	messages
	reliable	reliable
order guarantee	causal delivery	FIFO delivery

Figure 4.3: Properties of the dissemination protocols

4.5 The Causal Gossip Protocol

The first dissemination mechanism we present is *causal gossip*, a reconciliation protocol for mobile replication that pulls unknown events from other processes, independent of their origin. A participating node of a replication will usually try to stay up-to-date by periodically performing causal gossip exchanges with all nodes it can reach. It can find its peers with the help of discovery protocols that tell it when nodes become reachable or by periodically trying well-known addresses.

As each node of the replication system follow this process, updates to the replicated data eventually reach all nodes in the replication system. Furthermore, as the protocol ensures that only unknown changes are transferred, each node receives any change only once[1].

The application domain of our causal gossip protocol are mobile systems that locally update replicated data and are interested in learning changes that were made by remote processes. Mobile systems can particularly profit from gossip-style communication. As node do not only forward their own messages, but also events from other nodes, two devices A and C never have to communicate directly to exchange changes to the replicated data, but can do so over a common peer B.

4.5.1 Event Dissemination

A gossip protocol transfers all changes that are unknown to the remote party independent of their origin. It operates in two steps (see Fig. 4.4):

1. A target node A requests the latest events from a source node B by sending its current summary vector.

2. Node B then compares node A's summary vector with its own in order to compile the set of events unknown to node A. Node B extracts these events from its local storage and sends them back to the target node A.

[1]For a detailed discussion of epidemic change propagation and its properties please refer to [93].

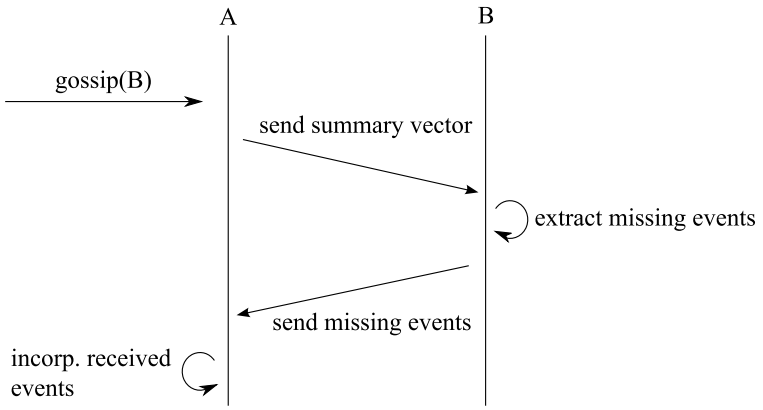


Figure 4.4: The basic gossip protocol requests all unknown events from a remote node.

When node A has received the remote events, it incorporates them into its data replica. Thus each node stores all local and foreign events that the node knows about and is able to update other nodes up to that point.

We extend the basic gossip protocol so that it efficiently disseminates full causal information:

1. We emit *reconciliation events* that fully capture causal relations.

2. We maintain causal order of events when we transfer them between hosts.

3. We maintain causal order when storing events in the local log.

Causal dependencies between nodes are only generated when nodes communicate. Furthermore, gossiping is effectively an information flow in only one direction. Thus we can capture all causal dependencies between hosts by recording a **reconciliation event** that records the process states of the two nodes participating in a gossip exchange. As with all events, a reconciliation event r has an event identifier $id(r)$ that names the state of the process when it initiated the reconciliation gossip and carries the state of the target process $target(r)$ in which it was reached by the reconciliation request. A reconciliation event is treated as a normal event by further reconciliation requests and thus disseminated to other nodes.

This extends the basic gossip protocol between A and B to (see Fig. 4.5):

1. A sends a reconciliation request to B.

2. B sends unknown events to A, *including its own current state* (B,t_B).

3. A generates a reconciliation event r with $target(r) = (B,t_B)$

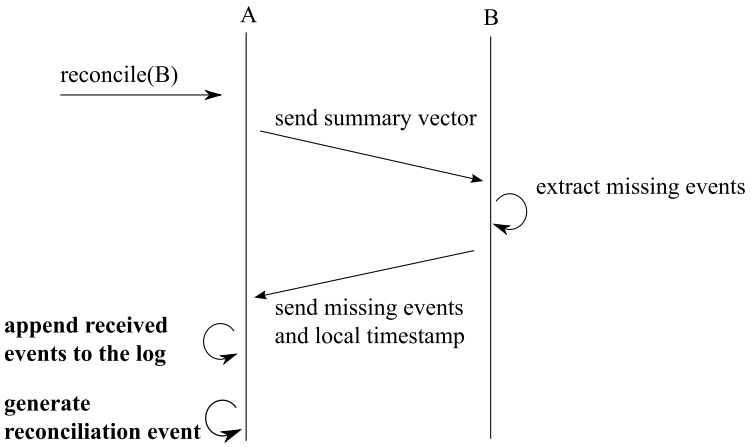


Figure 4.5: The causal gossip protocol extends basic gossip to preserve causal order and dependencies.

Properties

The causal gossip protocol implements causal delivery, so all of an event's causal predecessors are delivered before the event itself. That means that the summary vector of a process not only names the latest received events, but also implies a causal dependency to all their predecessors.

Because causal gossip implements causal delivery, a direct causal dependency implies that all its causal predecessors are named by the dependency vector of an event. The partial order of causality is then implemented with the dependency vector as follows:

$$e_1 \longrightarrow e_2 \Leftrightarrow dep(e_1)[process(e_2)] \geq time(e_2),$$

i.e. the source process of e_1 knew e_2 when it was created. It follows directly that

$$e_1 \parallel e_2 \Leftrightarrow \begin{array}{l} dep(e_1)[process(e_2)] < time(e_2) \quad and \\ dep(e_2)[process(e_1)] < time(e_1). \end{array}$$

4.5.2 Logging Events and Reconstructing Causal Dependencies

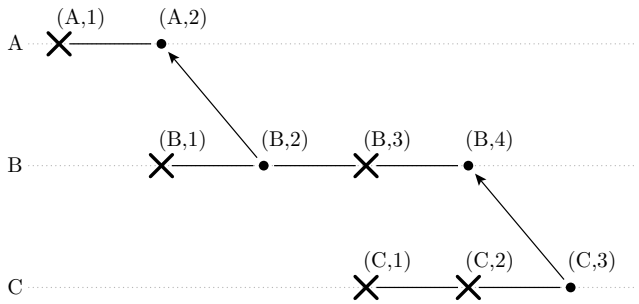
Causal gossip uses the shared log directly to retrieve local events and to store remote changes. It does so in a strict append-only manner: new local and remote events are always appended to the log as they are created or received, respectively. This way of logging events preserves causal order for an event e, because all its local and remote causal predecessors are present in the local log per definition when e is created.

Because causal gossip maintains causal order as events are transferred, the causal order between events is also preserved across nodes. This means that linear iteration over the log is always in causal order; an event is visited before any of its causal successors are visited. Concurrent events may be visited in any order.

Consequently, a causal relation between two events determines the order $<_L$ in which events are stored in the log:

$$e_1 \longrightarrow e_2 \Rightarrow e_1 >_L e_2$$

As an example, consider the following causality graph as seen by node C:



This graph will result in the following order of events in the log of node C:

$$(C,1)-(C,2)-(B,1)-(A,1)-(A,2)-(B,2\rightarrow A,2)-(B,3)-(B,4)-(C,3\rightarrow B,4)$$

In addition, the node logs the reconciliation event after a gossip exchange. This reconciliation event is appended to the log just after the events that the process received in the reconciliation response. With its help, we can compute an event's dependency vector $dep(e)$ in a recursive way. We start from the observation that each event e with $id(e) = (p,t)$ can have a maximum of two direct causal predecessors:

1. The first event recorded at a node has no causal predecessors.
2. A change event e with $id(e) = (p,t)$ has one direct causal predecessor e^{-1}, its local predecessor with the event id $(p,t-1)$.
3. A reconciliation event r with $id(r) = (p,t)$ has two direct causal predecessors, the event $(p,t-1)$ directly preceding it and the event named by the target state $target(r)$.

Thus the recursive formula for the dependency vector $dep(e)$ becomes:

$$dep(e) = \begin{cases} max(dep(e^{-1}), id(e)) & \text{for normal events} \\ max(dep(e^{-1}), id(e), dep(target(e))) & \text{for reconc. events} \end{cases}$$

with max being the component-wise maximum of vectors. Thus the log stores local and remote events along with their event identifier $id(e)$ and the event's dependency vector $dep(e)$ in a space-efficient manner.

The direct causal dependencies between events also induce a directed acyclic *causality graph*. The causality graph is initially empty, and is built with the local recording of reconciliation events and the reception of reconciliation events from remote nodes. Because these events are kept in the local log, the causality graph does not need to be maintained separately, but can also be generated on demand from the local log.

Log Pruning Issues

In order to calculate $dep(e)$, the log that causal gossip generates relies on information contained in older events. When those events would be removed from the log by a log pruning mechanism, some of the dependency vectors may not be fully computable. We will now define the notion of an **incomplete dependency vector** and use it to derive a definition of obsolescence from the perspective of calculating dependency vectors for causal gossip.

To that end, we extend the algebra for calculating dependency vectors. We introduce a state $\perp$ (*unknown*) in addition to the state values from $\mathbb{N}$.

Definition 4.7 *An incomplete dependency vector is a dependency vector with components* $\in \mathbb{N} \cup \perp$. *A* complete dependency *vector is an incomplete dependency vector with all components* $\neq \perp$.

An event whose predecessors have been erased from the log has a dependency vector with all components set to $\perp$ (*unknown*). We replace the *merge* function for two dependency vectors with the following merge function:

$$merge(a,b) = \begin{cases} \perp & \text{if } a[i] = \perp \wedge b[i] = \perp \\ a[i] & \text{if } a[i] \neq \perp \wedge b[i] = \perp \\ b[i] & \text{if } a[i] = \perp \wedge b[i] \neq \perp \\ \max(a[i], b[i]) & \text{otherwise} \end{cases}$$

Here we see that an $\perp$ (*unknown*) vector component can be eventually overridden by a *known* vector component. If we recursively start computing dependency vectors from the first events in the log, all but their "own" component of the vector will be $\perp$ (*unknown*). However, when the dependency vectors are merged to calculate dependency vectors of later events, the unknown components disappear and beyond some point the dependency vectors will become complete.

In order to get all complete dependency vectors for all necessary events, we have to adapt our definition of obsolescence for log pruning accordingly. Generally, using the all-know condition a^1 as the obsolescence criterion, we will be able to compute complete dependency vectors only for the latest events. If we use the a^2 instead, we will be able to compute dependency vectors for all events after the all-know cut defined by a^1.

4.5.3 Discussion

Because it can transfer changes over third parties for nodes that never communicate, the causal gossip protocol is particularly suited for use in mobile environments. Furthermore, it makes up for the lack of communication and storage resources that is commonly present in mobile devices by using these resources efficiently. It draws this efficiency from three properties of the mobile environment:

1. Reconciliation is not a continuous process but has to be invoked explicitly in order to request and import remote events. Therefore causal dependencies to remote events are only introduced at distinct points in time, and can be captured efficiently by special reconciliation events.

2. Events are sent by gossip in a way that preserves causal order. Thus an event's causal predecessors have been already received when the event itself is received. In consequence, we do not need to log the dependency vector of events, but simply their event identifier and payload.

3. Because reconciliation is a comparatively rare occurrence, a series of events from one source with dense logical timestamps can be created, which allows us to suppress header information for all but the first event of the series.

Taken together, these optimizations allow us to disseminate events with full causal dependency information without explicitly transferring this dependency information or even the full event identifier over the wire.

4.6 The Direct Send Protocol

4.6.1 Event Dissemination

Our second event dissemination protocol, direct send, is a reliable FIFO multicast protocol. Its target domain are replicated servers that can directly communicate with each other. We recapitulate reliable FIFO multicast here and those of its properties that are relevant for its use as a change event dissemination mechanism for a replication system.

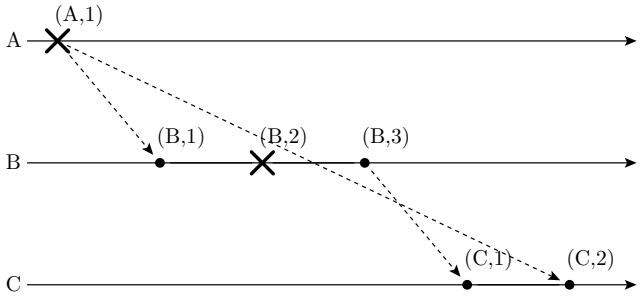
1. Processes immediately send new local events along with their event identifier and dependency vector to all other processes that are registered peers.

2. The receiving process orders the received events in the order of the sending process's logical clock and delivers them locally for further processing. Out-of-order events are buffered and duplicates are dropped.

3. When the receiving process detects a gap in the event order (one or more events are missing), it periodically sends a $NACK$ message to the sender until the gap is closed.

Properties

As a FIFO channel, direct send respects local causal order. However, it does not respect causal order between events of different processes. This fact becomes evident in the following scenario (taken from Sec. 4.4), in which the event $(B, 2)$ is received by C before its causal predecessor $(A, 1)$:



Fortunately, the operation model that we have posited in section 4.2 is not dependent on a causal dissemination facility. While event $(A,1)$ reaches host C late after its successor $(B,2)$, it does not necessarily contain any new information. When both change events modify the same data, $(B,2)$ already subsumes the changes $(A,1)$ made because it is causally dependent on it. Therefore the initial change in $(A,1)$ can be safely discarded. In turn, when both change events modify different data, there is no actual data dependency implied and the order of events does not matter.

The example shows a case where a naive interpretation of the dependency vector of $(C,1)$ would signal concurrency to event $(A,1)$, but in fact a causal dependency exists. We call this a **false concurrency**. This case can occur because direct send does not preserve causal order between events from different hosts, which means the transitivity in the dependency vector may be lost.

In order to be able to determine causal relations correctly, we have to reintroduce the transitivity of causality across processes. This requires that we recursively follow dependency vectors until we can rule out the possibility that further dependencies do not contain new information. We compute the true causal dependency vector $dep_{true}(e)$ by recursively gathering dependencies as contained in the events' vanilla dependency vectors $dep(e)$ and computing their maximum.

$$dep_{true}(e) = max_{e_d \in dep(e)}\ dep_{true}(e_d).$$

We can stop this in a recursion depth of $N_p - 2$ (with N_p being the number of processes in the system), because we only have N_p other processes and the causality of the last process can be checked by its potential successor.

They key property of the direct send protocol is that it **enables causal consistency for changes without an underlying causal delivery mechanism**. While causal gossip provided us with causal delivery for free, a non-gossip multicast protocol for online replication would need to take extra measures to be able to guarantee causal delivery: when it receives a message for which it misses any causal predecessor, it would have to delay the delivery of the message, resulting in longer message latencies and a strong coupling of the system. In contrast, direct send allows fast and direct delivery of messages in the non-loss case by using a FIFO delivery guarantee.

When causal delivery is violated by the reception order, causal consistency can be re-established by exploiting the semantics of our change tracking. A higher-level

algorithm, such as the one we describe in section 6.4, can detect any non-causal delivery by suppressing operations that arrive late, i.e. after their causal successors. It can do so because a previously delivered causal successor already contains the information in the suppress message. In our example, this is the case for operation $(A,1)$, which arrives after its causal successor $(B,2)$ at C. Being a causal successor however, its conveyed change is already contained in $(B,2)$ and therefore it can be suppressed in order to gain causal consistency.

4.6.2 Logging Events from Direct Send

As in the causal gossip protocol, direct send appends any received event directly to the log. When storage space is an issue, we can save some space by only storing those dependencies of an event that have changed from its local successor. When some replicas create events at a low rate, we can avoid re-storing their state in every event. When a log pruning mechanism deletes obsolete events, the same rules as for causal gossip apply: in order to be able to compute all dependency vectors for events after the all-know cut, we have to keep the log from a^2 on.

4.6.3 Discussion

Direct send is a reliable multicast protocol that directly disseminates changes along with their dependency vectors to all replicas. Its reliability mechanism integrates on the level of replica operation timestamps and does not need an underlying reliable transfer protocol such as TCP.

We have argued that direct send, despite its FIFO-only delivery guarantees, is able to support causal consistency of changes to the replicated data. Because we capture data updates as changes and not as operations, a higher-level mechanism can detect violations of causal delivery and suppress received changes in order to reestablish causal consistency. We have also demonstrated that FIFO delivery creates the problem of **false concurrency detection** and shown how to avert this problem.

The use of a FIFO delivery protocol has other important advantages. As a non-causal delivery protocol, the delivery of a change is not dependent on the reception of all its causal predecessors from other hosts. Any received change from one host can be delivered immediately and is not dependent on the progress of other replicas. On the system level, this implies a **loose coupling** between replicas in failures and performance.

Chapter 5

The Log as a Storage Mechanism

5.1 Overview

In previous chapters we introduced the log as a data structure that supports ordered storage and retrieval of events and operations for change recording and dissemination. The change tracking and dissemination algorithms of chapter 4 use the log in the following way:

- **write – append:** The log is only modified in an append-only fashion. Recording local changes results in an append operation of event and operation records. Change dissemination does not modify the log, but events and operations received are eventually imported by an append to the log. Note that using the log in append-only manner is important for performance as it has a high locality of access.

  In summary, the operations are:

  - `appendEvent(log,event)`
  - `appendOperation(log,operation)`

- **read – iterate:** Causal gossip accesses to the log by reversely iterating over it. When causal gossip retrieves requested events and their operations, it iterates backwards over the log until it has found all requested events. Dependency vectors are calculated in a similar way: their recursive computation results in a backwards iteration because the log is causally ordered. Direct send does not fetch single entries from the log, but accesses the log in a similar fashion when it helps remote nodes to catch up after a crash.

  In summary, the operations are:

  - `entry = getTail(log)`
  - `entry = getPredecessor(log,entry)`

The algorithms referred to the log as an abstract data structure – we did not consider any practical aspects of the log when it is implemented as part of a real

replicated storage system. In this chapter, we will close the gap between the abstract log and the underlying persistence primitives and provide principles and mechanisms that are important when the log is used as a persistent data structure that is part of a structured storage system. In particular we cover:

- *Logging and Consistency.* We introduce the concepts of interior and exterior consistency of replicated data and show how they can be maintained by unifying replication logging with write-ahead logging (WAL, ref. to [59, 86]) technique that is commonly used by database systems for recovery from crashes.
- *Log Garbage Collection.* Logs need to be compacted from time to time to preserve space. We show how obsolete records can be identified and how the log can be compacted.

5.2 The Log as a Database Log – Principles

We will first elicit important principles for maintaining consistency in a replication system by using the replication log as a recovery log and then present mechanisms that enforce these principles.

5.2.1 Consistency Aspects

A sine qua non requirement for a replicated storage system, as for any storage system, is that it must maintain the consistency of its entrusted data under all circumstances. Generally, the consistency of data is at stake when data that it is stored redundantly is changed. If the system crashes during a change, parts of the data might already reflect the new state while other parts contain the state before the change.

In a log-based replicated storage system, at least two forms of redundancy exist:

1. **Local redundancy**. Any data is stored both in the log as part of an operation and in the local database.
2. **Remote redundancy**. Any data is stored on all replicas.

These two forms of redundancy result in two consistency problems: interior consistency between the log and the local replicated state and exterior consistency between the replicas:

1. **Interior consistency** of the replicated data inside a replica. The storage system of a replica usually needs to store the local replica state with some redundancy. This can be the existing redundancy between the primary index of the data and the log, but can encompass more redundant storage when additional access paths (such as secondary indices) to the data are required.

2. **Exterior consistency** of replicated data between replicas. When one of the the replicas is changed, the change has to be reproduced at all its peers in way that allows the replicas to become consistent again eventually.

In order to ensure consistency even in the presence of crashes, any change to the replicated data must be an **atomic operation on both the local data replica and the local log** (see Sec. 5.3.1). For interior consistency, this guarantees that all information in the log is represented in the local data replica and vice versa. Exterior consistency is primarily the responsibility of the replication algorithm, however the replication algorithm alone cannot guarantee exterior consistency in all cases. This is because consistency between replicas is defined at the event level, but consistency of the data replicas is determined at operation level. Because an event can consist of more than one operation, the import of this event must be atomic – if it fails, the whole event can be retrieved again from a remote replica and imported as a whole, but only when the previous attempt is completely undone.

5.2.2 The Log as a Database Log

Consistency in the face of crashes can be maintained with the help of well-established database consistency techniques. Modern database systems keep a log of all changes that have been made to their data with enough information to undo and redo any transactions [86, 59]. This **database log** tracks changes similar to the replication log, and a hybrid log can be maintained.

The hybrid use of the log requires operations to be logged in a way that allows both their replay for recovery and for updating remote replicas. This requirement essentially implies that all information in the log is **replayable**:

- **Remote replayability** means that each replica has to store both its local operations and remote operations it received in a way that allows the replica to reproduce them in the form it has disseminated or received them. This is necessary to ensure that a new remote replica can be created and existing replicas can be continuously updated or catch up after they have crashed (replication log).

- **Local replayability** means that the log has to contain the necessary information in a way that is suitable for recreating the local data replica (database recovery log).

The main problem for local replayability is concurrent operations. Because they can arrive in any order and can thus be appended to the log in any order, the resulting stream of operations from the log as is would result in inconsistency between replicas. This problem must be solved without sacrificing remote replayability. Local replayability is not only important for the consistency of the local data replica after a crash, but has also performance implications. Local replayability allows a process to

execute the operation stream from the log directly on the local data replica without piping the stream through concurrency handling logic.

Apart from ensuring consistency, the unification of the replication and the database recovery log and the use of database recovery techniques can help **improve I/O performance** in three ways. First, we can avoid maintaining a separate recovery log for the local data replica and save its additional I/O operations. Second, with the transactional semantics of a database recovery, flushing data to the log can be deferred and done in batches, which improves general I/O throughput at the expense of delaying individual changes (**group commit**). Third, database recovery techniques allow the replicated database system to follow a NO-FORCE policy that only flushes changes to the log and defers persistent updates to the data (see [86, 59]).

5.3 The Log as a Database Log – Mechanisms

5.3.1 Atomic Updates of the Log

The use of the log as a recovery log allows us to make atomic changes to the log and the local data replica.

Consistency for local updates. In normal operation, both the local application and the replication algorithm modify the state of the local data and append operations to the log. Both the log and local database have to represent the same information.

It follows that any modification to either must be carried out as part of an **atomic operation** to both representations of the data. This can be achieved by using the general write-ahead logging protocol [86]:

1. Append the operation representing the change to the log.

2. Modify the local database accordingly.

3. Mark the logged operations as *committed*, either by appending another commit record or by re-writing the appended operation with an appropriate marker.

In addition to this update protocol, a recovering replica needs to re-execute any *committed* operations that are in the log but not in the data on startup (**redo mechanism**) [86]. Together, the protocol and the ability to redo changes guarantee that the log and the local data are always consistent. When the replica fails after step 1 or step 2, the operation will be re-executed on the local replica and the system will be consistent again. This is true because any of our logged operations as we have defined them in section 4.2 are **idempotent**: after they have been executed on the local data once, any further re-executions will not change the data further.

Consistency for remote import. Ensuring remote consistency requires dealing with the fact that a replica might crash while it is importing remote operations that it has just received. This event is not followed by data loss, because any lost information can be retrieved again from remote replicas. However, care must be taken that the replica re-establishes a state that allows it to retrieve and import the remote updates again. This consistent state is the state that the local replica had before attempting to update itself. Thus, any import of remote operations must also be done in a transactional manner with event granularity:

1. Handle the concurrent operations of an event.
2. Append the resulting operation stream to the log.
3. Mark the appended operations as *complete* (*transaction commit*).
4. Modify the local database accordingly.
5. Mark the block of operations as *committed*.

This protocol requires an additional logic on startup: when the system finds any operations in the log that are not marked as complete, it erases them from the log. If it finds complete, but uncommitted operations, it re-executes them on the local data as it is asked to do for maintaining local consistency.

5.3.2 Logical and Physical Log Order

We solve the problem of maintaining replayability for concurrent operations that may appear in any order by superimposing an effective **logical order** over the **physical order** of operations in the log. This logical order is a strict total order[1] between operations and determines in which order operations in the log need to be replayed for a consistent result without having to pipe the events through the concurrency handling algorithms.

We implement the logical order over physical order with two mechanisms: ordering by suppression and re-ordering by re-doing operations. Both mechanisms use the concept of **local operations** and **local operation modifiers**, which define additional operations or operations parameters. Local operations and operation modifiers are strictly local to a replica, which means that they are never exposed to other replicas, in order avoid violating the global replayability property. Apart from ordering and re-ordering, local operations are also used by the concurrency handling mechanisms that we present in chapter 6.

[1] Here we wish to stress that this is a total order in the mathematical sense and not in the distributed computing definition of the word.

Ordering Operations by Selective Suppression

When we receive operations in an online replication system, we need to append them immediately to the log and can not wait until all possible concurrent operations have arrived and the final effective order of operations is determined. The first mechanism, **ordering**, solves this problem of mapping a pre-defined logical order to the physical log order.

Consider two concurrent operations A and B that are required to have the logical order $B < A$. Two replicas may receive the operations as A,B or B,A and append them to the log. We can enforce the logical order by defining a local **suppression** bit that is part of each logged operation. The suppression bit tells the replica during replay whether an operation shall be executed on the local replica or not. When the operation is executed, it overwrites the effect of operations earlier in the log. If suppressed, the effect of earlier operations is maintained. Then the physical order $A, (B)$ (B suppressed) is equivalent to B,A and both represent the same logical order. Because it is a local modifier, the suppression bit is not communicated to other replicas.

Re-Ordering Operations with Redo

Ordering by suppression solves the logical ordering problem, if the logical order is known before the operations arrive. However, our algorithms sometimes determine the logical order while they are receiving events and therefore they must be able to re-order operations after they have been appended to the log in a certain logical order.

Re-ordering of operations can be implemented by appending the operation that is supposed to be the latest one according to the logical order as a local operation. This added operation is a local operation so that it has no effect on remote replicas. Consider three operations A, B, C that have been appended to the log in this (physical and logical) order. When an algorithm later decides that A should be after C in the logical order, it has to use the re-order primitive. It appends A as a local operation to the log. Re-ordered operations are local operations and are therefore not disseminated to other replicas.

5.4 Log Pruning and Compaction

In a log-based replication system, the log needs to be compacted periodically so that it does not grow without bounds. This process can be separated into two tasks: identifying and erasing obsolete records and reclaiming the unused space.

5.4.1 Identifying and Erasing Obsolete Records

The decision whether a log record – an event or an operation – is obsolete depends on how the log is accessed. Generally, the algorithms of chapter 4 and the concurrency

handling algorithms of chapter 6 access the log as a structure representing causal information. Because they adhere to this access pattern, the information they require can be **characterized by the all-know conditions**. Depending on the algorithm, we require events to be present that lie within the a^1 or the a^2 cut, respectively.

For event records, this obsolescence condition can be directly computed via the event records' dependency vector. The state of other record types is determined by their enclosing event record. In order to avoid computing this state for every encountered record, we exploit a property of the causal order of the log, namely that we can compute a safe lower bound for the respective $all-know$ conditions. This sliding **lower bound** is a **log offset** that can be compared with the log offset of a record in question.

All records that lie before the respective all-know cut or the all-know offset can be erased from the log. While the all-know cut defines a clear cut through the causality graph, its projection on the log will create a region in the log where only records of events from some of the hosts are obsolete. Erasing them creates gaps of free space in the log.

5.4.2 Reclaiming Space by Compaction

Erasing single records can create small gaps of free space in the log. This free space, however, can not be used to store new data because we use the log as an append-only data structure, and only store new data after the last record. Consequently, in order to create free space, gaps have to be closed by moving records towards the beginning of the log so that the last record is eventually moved.

Log compaction can be done as one cleanup run over the whole log, or iteratively by moving a batch of records and then suspending for some time. In the latter case, the system keeps an offset of where the latest batch of move steps has been suspended. This way, the log compaction can be done step-by-step as postprocessing of an application's operations on the database. When each operation of the application progresses the compaction one step further, the compaction cost is amortized over the operations that were executed during the compaction phase.

The database can keep track on how much gap space exists in the log. When too much free space is wasted this way and the database load is light, the database can initiate the cleanup unilaterally and defragment the log. For large databases, however, moving all records towards the beginning of the log can be a time-consuming task. A modification to the compaction algorithm can avoid long defragmentation times: instead of moving all records to the beginning of the log, the log is virtually divided into larger blocks whose boundaries act as compaction boundaries as long as enough records remain in the block. This way, a moved record within a block does not affect all records after it, but only those inside the boundary of the next block. The operating system can physically reclaim the space of empty blocks as a part of a sparse file. When the block size is large enough, a linearity of log records is still given, similar to the approach found in extent-based file systems (see for example [83]).

Chapter 6

Replica Consistency without Coordination: Handling Concurrent Operations

6.1 Overview

In the previous chapter we have shown how changes to replicated data can be logged and disseminated in a way that preserves data dependencies and thereby enables remote replicas to reproduce any modifications to the replicated data. In essence, each replica can determine the latest change operation in a chain of causal dependencies and execute it in order to generate that latest state of the replicated data. While these mechanisms are able to allow detection of concurrent changes, they can not enforce consistency when replicas have issued changes concurrently.

In this thesis we follow a weak-consistency approach to replication: instead of coordinating all attempts to change the replicated data, we accept all changes immediately and coordinate them later. Conceptually, this lack of coordination allows applications to issue writes concurrently. Concurrent changes have no data dependencies to each other and therefore introduce the possibility for the state of the replica to diverge: if concurrent changes address the same data, two or more **versions** of the same data co-exist after their execution in the same replication system.

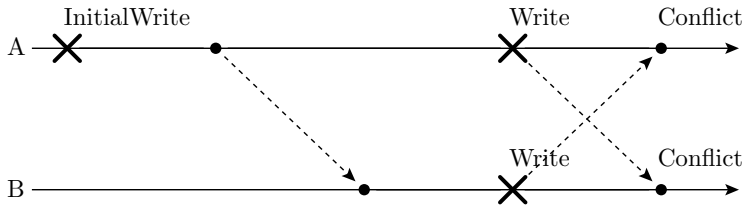


Figure 6.1: Concurrent operations on the same data are issued, disseminated and are detected as a conflict.

The example in Fig. 6.1 illustrates the concurrency handling problem. Process *A* creates a data item and disseminates it to *B*. Later, both *A* and *B* change the data concurrently. At this point, two different versions of the replicated data exist in the

replication system. Then the replicas send their change to their peers, where the concurrency and the conflict can be detected and handled.

When concurrent changes are disseminated to other replicas, the conflicting versions of data that they convey must be handled in a way that does not compromise consistency. To that end, we think of the alternative versions of the replicated data as **branches**, and see the act of concurrently changing the data as an implicit **branching** operation.

In this chapter we will present **Distributed Consistent Branching** and **Distributed Consistent Cutting**, two algorithms that are able to handle these concurrent operations in a way that leads to eventual consistency. The design of both algorithms was done in a **deductive** way. Given causal consistency and mechanisms to preserve it, we have sought algorithms that extend this premise in a natural way and provide applications with useful primitives. Hence, the design of both algorithms solely builds on the dissemination and persistence primitives of chapter 4 and 5 and extends them with concurrency handling primitives to a full replication framework that can be used by applications.

6.1.1 Distributed Consistent Branching: Branching and Merging

Our first solution to the concurrency handling problem, **Distributed Consistent Branching** (DCB, Section 6.3), detects the implicit creation of branches by concurrent changes and explicitly **recreates the branches on all replicas in consistent manner**. To that end, it builds on an extended data model that is aware of version branches of the replicated data.

These branches can handled eventually by a **merge operation**, which allows an automatic conflict resolving algorithm or a human operator to merge the concurrent versions back to a single main version. With this explicit and consistent recreation on all replicas, branches are available to applications and can be operated on: they can be explicitly addressed by write operations or merged together as a means of conflict resolution.

Branching and merging is similar to the well-known branching that can be found in revision control systems like CVS, SVN or git. Different than these more general branching mechanisms, our mechanism is restricted in a way that it models the implicit branching of concurrent operations in an explicit manner and does not require any other metadata than the causal information from the log.

Branching and merging assumes that conflicts can be handled by merging the version branches through a deterministic procedure or by a human operator. Such an environment can be found in Personal Information Management (PIM) or Computer-Supported Collaborative Work (CSCW) applications. The branching mechanism is particularly interesting for mobile systems, where periods of concurrency can last for a long time and therefore version branches can amass changes without restricting the availability of data.

6.1.2 Distributed Consistent Cutting: Cutting branches

Our second proposed mechanism for handling concurrent operations is **branch cutting**. Instead of explicitely recreating branches on all replicas, the **Distributed Consistent Cutting** (DCC, see Sec. 6.4) algorithm **dynamically selects an effective branch** and drops all other branches.

The application can influence this process by assigning a priority measure to operations. At any point of time, the data set contains only the branch that is considered the current main branch so that all further operations are causally dependent on it.

Branch cutting is a viable solution for replication systems where the data semantics do not allow for explicit branching, where consistency of the data is more important than single changes, or where no human operator is involved that could investigate and merge branches later.

Because branch cutting provides consistency without coordination, replication systems in high-latency environments will profit from its use. This comes at the cost that branch cutting selectively drops changes to the data and therefore loses them. However, as we will see, the probability for loss depends on the amount of conflicting concurrency in the system and its communication latencies. Thus, the chance for change loss is small in systems that have a small to medium communication delay and a wide distribution of changed data.

6.2 Model of Operation

Generally, the algorithms that we present in this chapter are **event handlers** that handle remote events and the change operations they contain. They represent an integral part of the **import** process for a remote event and its operations:

1. receive an event,
2. detect and handle concurrent operations,
3. append it to the log and apply it to the local data replica for persistence.

While a replica may receive more than one event in a batch, the events are imported one after the other in causal order. Please note that this import process is only applied to remote events, local change operations are directly executed on the local data replica and appended to the log.

Concurrent events are handled in a way that leads to an eventually consistent state on all replicas. Our concurrency handling algorithm exercise control over the execution of the remote operation on the local data replica by transforming the operation stream of an event with the mechanisms that were presented in section 5.3.2. With the help of these mechanisms, they can order and reorder remote operations through local modifiers and can insert local operations in the operation stream of the remote event, thereby respecting the **replayability** of the operation stream. In

their operation, concurrency handlers must also respect causal order so that all data dependencies of non-concurrent operations are preserved.

From an algorithmic point of view, the algorithms that we present in this chapter are **online algorithms**. They handle operations as they are received and produce intermediate states until they converge to a final consistent state. Intermediate states depend on the set of concurrent operations that a replica has received to that time. These sets can differ across replicas and therefore intermediate states can be inconsistent across replicas. Because all replicas eventually receive the same operations, they will converge to the same consistent state. The reactive design of our algorithms also trivially guarantees their **liveness property**: because they do not have any internal state, they always make progress and can not dead-lock.

6.3 Distributed Consistent Branching

In this section, we present the Distributed Consistent Branching (DCB) algorithm that de-conflicts concurrent changes to replicated data by explicitly creating version branches [63]. Together with a merge operation, the algorithm is a mechanism that allows conflicts to be handled asynchronously.

We will first give a syntactical definition of the branched data set and operations on it, and then explain how DCB operates on this data model in a way that guarantees eventual consistency of the replicated data.

6.3.1 Operations on Branched Data

We have postulated that applications of our replication framework work on a dictionary data model of unique keys and their associated values (see Sec. 3.2.2). In order to account for the explicit branching of DCB, we extend the dictionary to allow multiple branches of a key's data:

key	branch name	data/value
k_1	b_1	...
	b_2	...
k_2	b_3	...

We use event identifiers to identify branches:

Definition 6.1 *A **branch name** b is an event identifier (p, t); we write $p-t$. A **key branch** is a tuple (k,b) of the key k and the branch name b. We write $k[b]$ for the branch name b of the key k. A **branched data set** D is a mapping $D : (K,B) \to V$ of (key, branch name) tuples to values.*

The application may change the local branched data set by assigning a new value to a key branch: $k[b] = v$. In response to this change, the replication system will log an **operation** that is transmitted to other replicas as part of an event.

Definition 6.2 *An **operation** $o(k[b],v)$ sets the value of the key branch $k[b]$ to v: $k[b] = v$. The **write set** $W(e)$ of an event e is the set of all key branches $k[b]$ of operations in e.*

While operations in the log always address a key branch $k[b]$ explicitly, applications may operate on plain keys k if there is only one branch of the key. The relationship between application changes to the data set and the operations that the replication system logs in response are:

1. If a key k is not in D, the application may create it by writing to k. The write results in the creation of the **default branch** of k. The default branch is named by the current local event identifier (p,t) (process-id and current local logical clock timestamp).

2. If D only contains one key branch $k[b]$ for k (the default branch), the application can implicitly address $k[b]$ by simply writing to k: $k = v$.

3. If D contains multiple branches for a key k, the application must address a key branch $k[b]$ explicitly. This way the data is always available for changes even in case of concurrency.

This results in the following programming interface:

action	branches of k in D	application change	logged operation
key creation	empty	$k = v$	$o(k[(p,t)],v)$
change	b_1	$k = v$	$o(k[b_1],v)$
change	b_1, ..	$k = v$	*failure: is branched*
change	b_1, ..	$k[b_1] = v$	$o(k[b_1],v)$
change	b_1, b_2	$k[b_3] = v$	*failure: does not exist*

The interface semantics make sure that an application does not need to care about the fact that it works on a branched data model as long as there are no concurrent changes. Without any branches (created in response to concurrent changes) all changes may implicitly address the default branch implicitly. When DCB detects concurrency, it will create and adapt branches accordingly, and the application will have to explicitly address the branch it wants to operate on.

Our semantics also imply that an application may only create a key branch when it creates the key with an initial change and as otherwise to address key branches that are already in the data set. These branches are created by the DCB algorithm in response to concurrent operations with the help of two operations:

- Via a local **rename** operation that renames a branch b_1 to b_2: $rename(k[b_1],b_2)$. Rename operations are normal operations that are internally issued by the replication system and are logged as such. However, as they are *local* operations and are not transmitted to other replicas (see 5.3.2) and are only effective at their originating node.

- A local **redirection** modifier $\{b_n\}$ that redirects an operation $o(k[b_o], v)\{b_n\}$ to modify the new branch b_n instead of the original branch b_o. If b_n does not exist, it will be created by the execution of the operation. Local modifiers are attached to remote operations when they are written to the log after their reception. They are not transmitted to other replicas when reconciling. Thus other replicas receive the vanilla operation $o(k[b_o], v)$ and will establish their own internal redirections.

When concurrent operations on a key branch $k[b]$ are received, DCB creates new key branches $k[b_1], k[b_2], ...$ that replace the original key branch $k[b]$ with the help of the local rename operation and the local redirection modifier. The branch that such an imported operation modifies is its **effective branch**:

- An change operation $o(k[b_o], v)$ followed by **rename** operation $rename(k[b_o], b_n)$ is effectively operating on the key branch $k[b_n]$.
- A redirected operation $o(k[b_o], v)\{b_n\}$ addresses effectively the key branch $k[b_n]$.

6.3.2 Causality of Operations on Branched Data

Changes to a branched data set are subject to the change dissemination and logging mechanisms and protocols of chapters 4 and 5. This implies that:

- Changes on one process are appended to the log of the process in their causal order. When there was no communication with other replicas between two changes, the changes share a common logical timestamp. Then they are part of the same event, transmitted together during reconciliation and have the same causal dependencies to changes contained in other events.
- Changes from different nodes are part of different events. The causal dependency between these changes is equivalent to the causal dependency between their events. Thus two operations are concurrent of their events are concurrent.

One implication is that each version of the data in the branched data set can be identified by an event:

Definition 6.3 *A **version** of a key branch $k[b]$ is the event e that contains an operation that writes on $k[b]$: $k[b] \in W(e)$.*

The identity between events and their contained operations allows us to argue about the causality of changes at the level of events. From the local log of a process, we can build the directed, acyclic causality graph of all events and their operations and compute a subgraph for each key branch $k[b]$. The nodes of this subgraph are events that contain an operation $o(k[b], v)$ that addresses $k[b]$, the edges are the causal dependencies between the events.

We call this subgraph the branch tree of $k[b]$ because:

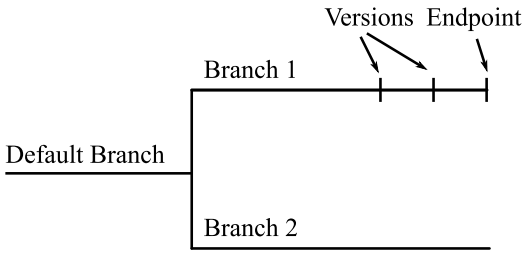


Figure 6.2: Branches, versions and endpoint.

- Its **root** is the event with the initial write operation that created the key. All subsequent versions of $k[b]$ form the nodes of the tree.
- Each node of the tree can only have **one parent**.

Each node can only have one parent because if it had two parents, it would have two causal predecessors that have changed $k[b]$ concurrently. However, two concurrent operations on $k[b]$ would have been detected as concurrent by DCB (as will see later) and redirected/renamed to new branches. Therefore the original branch $k[b]$ would no longer be in the data set, and thus there can be no subsequent application changes that address $k[b]$.

Definition 6.4 *The **branch tree** $B(k[b])$ of a key branch $k[b]$ is the subgraph of the causality graph C of all versions of $k[b]$.*

We call the leaves of the branch tree **endpoints**. Endpoints are versions in $B(k[b])$ that are maxima of the causal order relation (see Fig. 6.2). Because endpoints of the current branch tree of $k[b]$ contain the latest writes on a key branch, they are identical to the branches of $k[b_1], k[b_2], ..$ in the data set, which have been created by DCB in response to their concurrency.

Definition 6.5 *An **endpoint** (latest write) of a key branch $k[b]$ is a version that has no causal successors with operations on $k[b]$. For an endpoint e_l: $\nexists\ e_c$ with $k[b] \in W(e_c)$ and $e_c \longrightarrow e_l$.*

The branch tree of a key branch $k[b]$ captures all effects of operations that have directly addressed $k[b]$. In case of concurrency DCB will rename this branch and redirect operations and create new branches b_1, b_2, .., but the concurrent operations in the log still address $k[b]$ directly. Only renames and redirects ensure that they modify the effective branches and not their original branch $k[b]$. An application that finds these new branches in the data set after concurrency has been detected may change them. These changes and any subsequent changes will address one of the new branches b_1, b_2, .. directly and are therefore associated with their own branch tree and subject to their own concurrency handling process.

The branch tree of key branch can be **iteratively built** by a process as it receives new operations on $k[b]$. As all processes eventually receive the same operations on $k[b]$, they will eventually build the same branch trees for $k[b]$. We will exploit this fact by using the first version on a branch as the name for the branch. Branches are therefore named after the events that led to their (implicit or explicit) creation: when a branch is created by an operation in event $(A,5)$ it implicitly creates the branch with the name $A-5$. We can now define:

Definition 6.6 *A **branch** $\bar{b}$ of a key branch $k[b]$ for endpoint e_b is a subgraph of the branch tree T of $k[b]$ of all its events that are causal predecessors to e_b and not to any other endpoint of $k[b]$. A branch $\bar{b}$ is named after the event identifier of its minimal (earliest) element in causal order (**branch naming invariant**).*

We can derive the following properties from these definitions.

Theorem 6.1 *All branch names of a key k in a data set D are mutally causally concurrent.*

Proof Consider two branch names $k[b_1]$ and $k[b_2]$ with $b_1 \longrightarrow b_2$. For both branches there must be endpoints in the causality graph with operations on a common branch $k[b]$. Because endpoints are concurrent per definition, the corresponding branch names must be concurrent, too. $\diamondsuit$

Theorem 6.2 *A data set can contain at most $|P|$ branches, with $|P|$ being the number of processes in the replication system.*

Proof Consider a data set with $|P|+1$ branches for a key k. Because branches are identified by event identifiers, two branch names would have the same host component. This implies that these branches are causally related, which we have just shown is not possible. $\diamondsuit$

6.3.3 Distributed Consistent Branching

In the following we will show how DCB imports remote operations in a branched data set yielding an eventually consistent set of branches across all replicas. The algorithm computes the effective branches of operations and applies them to the local data set with the help of local rename operations and local redirection modifiers.

Distributed Consistent Branching is an online algorithm, which takes an received event as its input that has been received via the causal gossip protocol (Sec. 4.5). The algorithm operations in two steps:

1. It detects whether the operations belongs to an existing branch or defines a new branch. In the former case, it is redirected to the existing branch, in the latter the name of the new branch is computed (which is its event identifier) and the operation is redirected to this new branch.

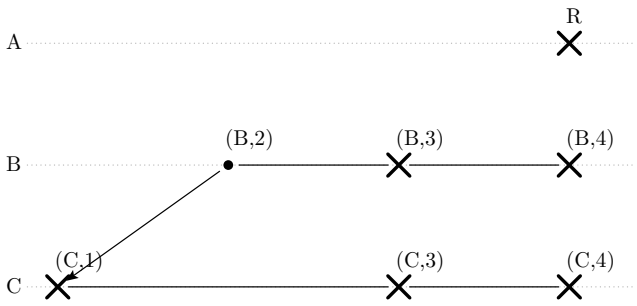
2. It detects whether any of the existing branches need to be renamed to comply to the branch naming definition (first operation on a branch) and uses a local rename operation to do so.

The definitions of the previous suggest an algorithm that uses the causality graph and branch trees to find and name branches. In order to be able to avoid working on graphs in the normal case, we annotate each branch in the data set with the latest event event that modified it (which is an endpoint):

key	branch name	data/value	*endpoint*
k_1	b_1	...	e_1
	b_2	...	e_5
k_2	b_3	...	e_3

We will support the presentation of our algorithm by following its steps in an example:

Example. We observe a process while importing a remote event R from node A.



The process has already imported remote writes from B and C and created branches from them in its data set (we mark events with dots and events that write to the key branch in question with a cross). This resulted in the following data set:

Key	Branch	Value
..	...	...
K	B-3	..
K	C-3	..
...	...	...

The causal dependency of event R is left unspecified and will be used to generate different cases later.

Finding a branch for the remote operation

When importing an operation $o(k[b_o], v)$ from event e_r, we can face two cases:

1. There is no key k in D. Then o is the initial write operation and can be directly executed.

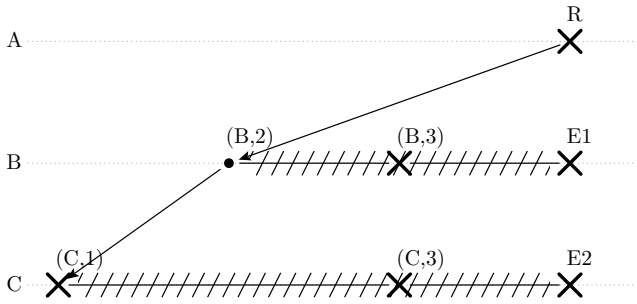
2. There are one or several branches for k in D. By comparing the dependency vectors of the annotated endpoints in D with the dependency vector of the remote event, DCB can find out whether the remote operation is concurrent to the latest operation that modified the branch (**concurrent endpoint**) or whether it is a causal successor (**non-concurrent endpoint**).

All events designated by endpoints and the remote event are versions in the same branch tree $k[b]$. Thus, **there can be at most one non-concurrent endpoint** for $o(k[b_o], v)$ in D. To identify the branch the remote operation belongs to, we iterate over all branches of $k[b]$ in D (Alg. 1).

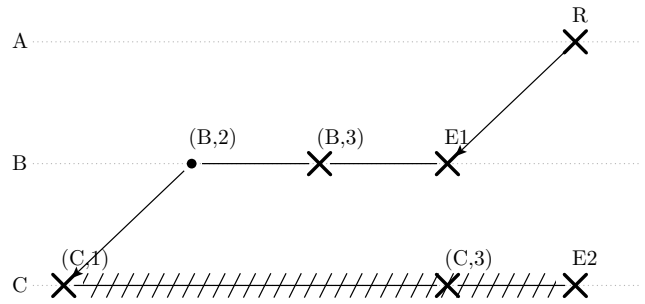
When we find **a non-concurrent endpoint** of a branch b_n, we know that the remote operation causally depends on it and therefore was referring to the same branch as the endpoint. This means for the consistency of the data that the replica has already received events that belong to a particular branch and thus previous runs of DCB have already created the necessary branches. Thus, the operation must be applied to the same branch of the data. We achieve this by redirecting the operation to this branch via a local operation modifier $\{b_n\}$.

If there are **only concurrent endpoints** in the data set, all latest writes to the data are concurrent to the remote event. Therefore the remote event must be the first event on a branch and thus defines the name of the remote branch. When importing the remote event, we establish a redirection to the new branch with the name of the remote event $\{e_r\}$.

Example. In our example there are two branches in the data set, $B-3$ and $C-3$, which have been created by an earlier execution of the algorithm when the corresponding events were received. Assuming a causal dependency of R to $(B,2)$, we have hatched the events that are concurrent to R. In this case we have two concurrent endpoints E1 and E2. Therefore R defines a new branch and its operations has to be redirected to $k[R]$:



If we assume a causal dependency from R to E1, endpoint E1 is the non-concurrent endpoint and R lies on its branch. Hence the operation has to be redirected to change the value of k[$B-3$].



Algorithm 1: *computeEndpoints*

Data: remote event e_r with operation $o(k[b_o], v)$, $D(k)$, the branches of k in D
Result: n non-concurrent endpoint, E_c set of concurrent endpoints

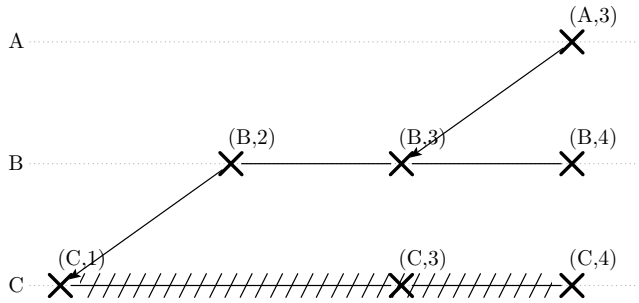
```
1 begin
2     E_c = ∅;
3     forall branches,endpoints (b, e_e) ∈ D(k) do        // gather endpoints
4         if e_e ∥ e_r then                               // b has concurrent endpoint
5             E_c ← (b, e_e);
6         end
7     end
8     return D(k) \ E_c, E_c
9 end
```

Adapting branches

Maintaining eventually consistent branch names is not straight-forward because concurrent events can arrive in any relative order. Concurrent events arrive one-by-one, possibly over longer periods of time, but are always imported instantaneously and their respective branches are created. Because the order of arrival likely differs between replicas, branches that have already been created might need to be adapted when new concurrent writes arrive.

After creating the branch for the remote event, we have to revisit the existing branches, check whether they still comply with the branch naming invariant and adapt their name if necessary. The following example illustrates this case:

Example. Consider the following causality graph, with the endpoint $(C,4)$ and its branch $C-3$ and three further concurrent writes $(B,3)$, $(B,4)$ and $(A,3)$. When it was received, $(B,3)$ created a branch $B-3$.



Because of their concurrency, $(B,4)$ and $(A,3)$ can be received in any order. When $(B,4)$ is received before $(A,3)$ it lies on the branch $B-3$, and the reception of $(A,3)$ would trigger the creation of a new branch $A-3$.
However, when $(A,3)$ is received first, it lies on branch $B-3$ (non-concurrent endpoint) and later $(B,4)$ would be redirected to its own branch $B-4$, resulting in an inconsistent branch naming.
In either case, one of the branch names does not comply with the branch naming definition any more, because the branch name $B-3$ has two causal successors, $(B,4)$ and $(A,3)$. It follows that the creation of new branches can require DCB to adapt names of existing branches.

In order find out which branches need to be adapted, we compare the branch names for k in D with the remote event. We call branches that are concurrent to the remote event **concurrent branches**. Branches that are causally related to the remote event are **non-concurrent branches**. Again, there can only be one non-concurrent branch, because branch names are versions and the remote event can only have one causally preceding version.

For the endpoints of the **concurrent branches** we already have the right branch names, because concurrent branches branch off before the point the remote event branches off and are therefore not affected. Thus we only have to compute a new branch name for a possibly existing non-concurrent branch, because its branch name is no longer complying to the definition of a branch name, i.e. the branch is named after its first exclusive version.

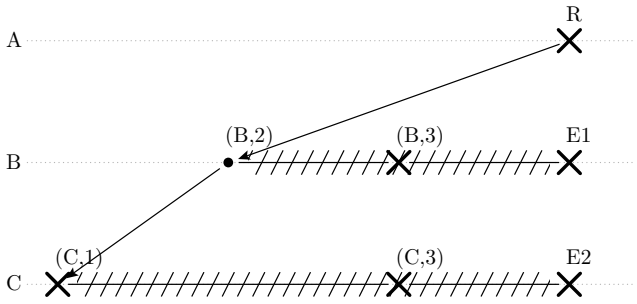
We compute the new branch name for the **non-concurrent branch** with the help of the branch tree. We start from the endpoint of the non-concurrent branch and look for the earliest of its causal predecessors that is concurrent to the remote event (and is therefore suitable as a branch name, see Alg. 2). We use this event's name as the new branch name and rename the branch in the local data set accordingly.

Algorithm 2: *computeBranchName* - Computation of the new branch name for the non-concurrent branch.

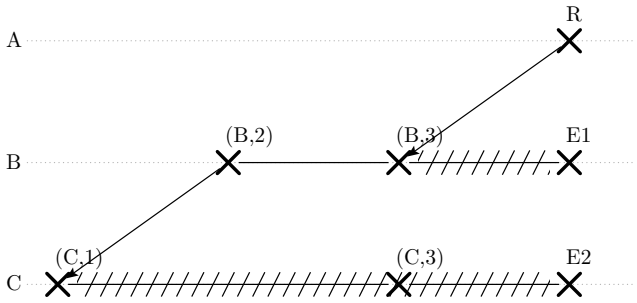
Data: non-concurrent branch to rename b_{nc}, remote event e_r, branch tree $B(k[b])$
Result: b_n new branch name

```
1 begin
2     b_n = b_nc ;
3     forall e ∈ B(k[b]) do        // find earliest concurrent predecessor
4         if b_nc ⟶ e and e_b ∥ e_r and b_n ⟶ e then
5             b_n = e;
6         end
7     end
8     return b_n
9 end
```

Example. First consider the following scenario, with the endpoint E1 and its branch $B-3$ and endpoint E2 and its branch $C-3$.
The concurrency set of R includes both the branch of E1 and E2. Thus we have only concurrent branches and no branch name needs to be adapted.



Now consider a scenario, where the branch $B-3$ of the endpoint E1 is not concurrent to R:



Then the name of the branch $B-3$ has to be adapted, because it is the causal predecessor of R and E1 and therefore no longer complies to the branch naming invariant. In our example, the new name of the branch is E1.

Distributed Consistent Branching

Together the presented algorithms result in the Distributed Consistent Algorithm as shown as algorithm 3. When handling operations from a received event, we first check whether the operations creates a new key (lines 2–5).

If the operation modifies an existing key, we compute the the non-concurrent and concurrent endpoints (line 6). If a non-concurrent endpoint exist, we redirect the remote operation to its branch (lines 7–8).

Otherwise the operations lies on a new branch that is named after its event (line 10). We check if any of the existing branches is a non-concurrent branch and adapt its branch name accordingly (lines 12–16).

Algorithm 3: DCB – importRemoteEvent($o(k[b], v)$)

Data: $o(k[b], v)$ operation in event e_r, D local data set, C causality graph
Result:

```
1  begin
2      if D[k] = ∅ then                                   // no branches exist
3          emit o(k[b], v) (create initial branch k[b] = v);
4          return
5      end
6      e_n, E_c = computeEndpoints(k[b],D) – Algorithm 1;
7      if e_n (non-concurrent endpoint exists) then        // branch exists
8          emit o(k[b], v){e_n} (redirect k[e_n] = v);
9      else                                               // create new branch
10         emit o(k[b], v){e_r} (redirect k[e_r] = v);
11         compute non-concurrent branch (if any);
12         b_nc = b ∈ D(k) : e_r ⟶ b;
13         if b_nc ≠ ∅ then                    // non-concurrent branch exists
14             compute endpoint of non-concurrent branch;
15             b'_n = computeBranchName(b_nc, e_r, C) – Algorithm 2;
16             emit rename(k[b_n], b'_n);
17         end
18     end
19 end
```

Correctness Argument

The correctness argument for DCB, as for any distributed algorithm, is separated into **liveness** and **safety** arguments. Liveness guarantees that the algorithm will eventually do something and does not dead-lock. In the context of DCB, this means that all branches are eventually created. Safety guarantees that the algorithm will always produce states which comply to its specification.

DCB is an event-driven, **stateless** algorithm and thus can not dead-lock per se. It is always in the same reactive state: import remote event. It has no other state than the local log and the local data set, both which can be said to be mode-less in the sense that they do not contain "special" states that influence the operation of the algorithm.

The safety requirement for DCB is that it must, for operations that operate on a particular key branch, **produce the same branches with the same values, independent of the reception order**. Because DCB relies on causal dissemination primitives, only the reception order of concurrent operations can differ.

When replicas receive the same set of operations, they will derive the same causality graph (see also [37]). For operations on a key branch $k[b]$ they derive therefore the same branch tree. The name of branches is unambiguously defined by the first operations on each of the branches (invariant). Thus the safety of DCB is equivalent to the fact that DCB (a) creates the branches independent of reception order and (b) operations are redirected/renamed to the proper branch. Property (a) is given because branches are either named directly after their first exclusive event (new branches) and are later adapted (Alg. 2). Property (b) is given because DCB always identifies the one branch whose endpoint is the direct causal predecessor of the remote operation.

6.3.4 Conflict resolution by Merging

The dual operation of DCB's implicit branch creation mechanism is the explicit merge operation. The merge operation allows conflict handling by merging branches back to a new default branch. We discuss both manual merging (by a user) and automatic merging (by automatic conflict handlers).

Manual Merging with the Merge Operation

Manual merging through an user interaction makes the merging process inherently indeterministic, and therefore the merging mechanism must be prepared to handle concurrent inconsistent merges. In addition, manual merging by a user is typically executed on a single replica and needs to be recreated on all other replicas. Thus, the merged branch only becomes available when causal gossip has disseminated it to all nodes.

Manual merging is supported by an anonymous merge operation, which contains the merged data and names the key branches to be merged. The anonymous merge does not explicitly name the merged branch, and takes the event identifier of the operation as the name of the merged branch. This ensures that concurrent merges would result in distinct branches.

The merge operation MERGE(*branch*, *data*) that merges all the branches of the default branch *branch* has the following semantics:

1. it creates a branch b that has the name of the event of the merge operation and contains *data*

2. it deletes all implicit branches of *branch*, independent of whether they were known when issuing the MERGE.

This implies that the MERGE operation also replaces branches that are yet to created as well as branches that are still written to implicitly. An application that wants to avoid that behavior has to restrict the usage of MERGE to branched keys whose branches are known to all replicas.

The handling of a local MERGE is straightforward: we delete all branches of *branch* and create a new branch with the current host-state as identifier and *data* as data.

Remote operations, including a remote MERGE must be imported in a way that adheres to merging semantics. Basically our definition says that any concurrent change needs to be ignored. We can implement this with the suppression mechanism. Concurrent changes that address a branch different of the default branch are non-conflicting and therefore do not need any particular treatment. Concurrently issued MERGE to the same branch create a branch with a different name and are therefore also not in conflict. This leads us to Alg. 4.

Algorithm 4: *importRemoteEvent*(o(k[b],v)) for operation o on key branch $k[b]$ from event e

```
1  begin
2      if o is merge operation then
3          delete all branches;
4          set k[b] = data;
5      else if o is write operation then
6          if concurrency set of e has a merge operation on b then
7              suppress o;
8          end
9          execute DCB;
10         importRemoteEvent(o(k[b],v));
11     end
12 end
```

Automatic Merging with the MergeTo Operation

Automatic merging procedures can by deployed when manual merging by a user is not feasible or necessary. Automatic conflict resolvers can implement general or application-dependent consistency policies for canceling or merging changes. Alternatively, they may simply commit the results of write operations when all nodes have acknowledged their reception. Automatic merging should be deterministic across nodes, so that a certain set of branch values always results in the same merged value.

Similar to MERGE automatic resolvers use a named merge operation that explicitly describes the branch of the merged data. Because the resolvers are deterministic, the named merge operation is not subject to conflict detection and resolution. Each automatic resolver has an assigned identifier that is incorporated into the merged branch name in place of the node identifier in order to ensure that all merges result in the same branch.

This MERGETO operation has the same semantic as the MERGE but uses an explicit branch identifier as the merge target. This branch identifier is not taken from the node name namespace but uses a separate name that identifies the branch to be generated by an automatic resolver. Concurrent operations on the merged branch are handled as for MERGE.

Concurrent MERGETO target the same branch and can be result in any order in the log. When the concurrent merges are concurrently written, normal DCB rules for concurrent operations on the same branch apply, despite the operations do not have a common causal root.

6.4 Distributed Consistent Cutting

In this section we present the *Distributed Consistent Cutting (DCC)* algorithm that guarantees eventual consistency in face of concurrent changes. Just as DCB, Distributed Consistent Cutting is based on the observation that any conflicting changes implicitly create a branch. Other than DCB, which reproduces this branching explicitly on all replicas, DCC selects only one of the branches to be present on all replicas in a consistent manner. For operations on this selected branch, it ensures total causal order. All other branches are *cut* and the changes of operations that lie on them are lost.

We will continue with the presentation of the DCC algorithm in three parts:

1. We explain the *rationale* behind the design of DCC and discuss inherent *trade-offs* and its use cases. We further give a detailed *specification* of DCC and argue that it guarantees eventual consistency.

2. We present a *general version* of the DCC algorithm and *refine* it for use in mobile replication systems that use causal gossip and online replication systems that use direct send.

3. We discuss failure handling and crash recovery.

6.4.1 Rationale

While DCB solves the concurrency problem of causal consistency, it only serves the limited class of applications that can adopt a branched data model. The majority of applications is designed with an unbranched data model in mind and assumes operation semantics that resemble those of an unreplicated database.

For applications of this class, we have developed the DCC algorithm, a lightweight concurrency handling mechanism that builds on our efficient causal dissemination and logging primitives. Because it is an asynchronous replication algorithm, the performance of replicas is decoupled from the performance of their peers and from the latency of the network. This can be particularly interesting in wide area network setups. DCC is able to reach this goal by allowing, in case of concurrency, the selective loss of updates and restricting the availability of data for changes.

Synchronous change coordination algorithms impose their costs even when concurrent conflicting changes are rare or can occur only under special circumstances. The motivation for the design of DCC is the observation that **concurrency can actually be a rare or controllable event** and not always worth the effort and disadvantages of a synchronous replication algorithm. When applications operate in such an environment and can tolerate the occasional loss of updates, the use of DCC can be a viable alternative to synchronous coordination algorithms. We will provide examples for such environments in the next section.

In order to be in conflict, changes must be concurrent in a causal sense and must modify the same data. Thus conflicting changes can only occur if two replicas **modify the same data within network latency** (and event processing time). After a change has reached another replica, concurrency with this replica is no longer possible because all subsequent changes to the data are causally dependent. Hence, DCC's branch cutting logic only needs to step in when two replicas update the same data time within the short window that is opened by the round-trip time, which we call the window of concurrency.

In an online replication system, this window of concurrency is usually in the order of 0.1 ms for LANs and goes up to about 100 ms for WANs. Thus, applications have to update the same data within this short period of time to be subject to DCC's branch selection. The frequency of updates to the same data depends on the application-specific change rate and change distribution. However, if the application is not implementing a system that provides coordination semantics between clients, the probability of a change to the same data within the round-trip time should relatively low. We will investigate the effect of network latency, the size of the data set and the number of replicas on DCC's performance in section 7.2.

When concurrency still occurs, DCC will react with the selective drop of changes. Thus, an application that choses to use DCC for handling concurrent changes must be comfortable with the possible **occasional loss of updates**. This could be an application for which it is either generally acceptable to lose updates or it may just be able to tolerate loss if the loss occurs rarely. In either case, DCC's branch selection logic is controllable by the application by associating operations with priorities.

6.4.2 Use Cases

Generally, those storage systems can benefit from weak-consistency replication techniques such as DCC's, which are concerned about availability, performance and scalability, but have no strong consistency semantics. Within this spectrum, DCC

provides causal consistency with low implementation complexity in the general case, but may need to select among concurrent updates in the case of concurrency.

Mobile replication. For mobile replication scenarios that require some degree of data availability, there is no practical alternative to weak-consistency replication. While DCB provides a general solution for this case, its explicit branching mechanism can be inconvenient and merge procedures too heavy-weight. In this case, DCC can provide simple and efficient solution: for data such as uncritical configuration data or measurement data, it can be more convenient if in case of concurrency the latest update wins and others are dropped.

For online replication systems, we envision several usage scenarios for DCC's consistency guarantees:

Alternative to Last-Writer-Wins. For storage systems with weak consistency demands, **Last-Writer-Wins** (LWW) represents a simple but effective solution to guarantee eventual replica consistency. Last-writer-wins replication assigns each change a timestamp from the real-time clock and orders changes at each replica according to this timestamps. While its implementation is straight-forward and light-weight, LWW is not able to preserve causal relationships between changes (see also Sec. 2.2.1).

As we will see, DCC is of similar implementation complexity as LWW, but is able to preserve data dependencies between changes and handles concurrent changes in a well-defined manner. An example for storage systems, which use LWW consistency but could benefit from DCC are directory services that experience high query rates and have a high availability demands, but have usually relaxed consistency requirements and low levels of update concurrency.

External sequencer. In distributed file systems, such as XtreemFS [64], the metadata server is usually separated from storage servers that store file data. However, some of the modifications of file data on storage servers requires updates to file metadata, most noticeably changes of the file's size and access time. In a parallel system, updates of this metadata can reach one of the metadata server replicas in any order and also concurrently.

Fortunately, POSIX requires changes to file data to be in total order and therefore the metadata updates that result of any file change can be tagged by sequence numbers that are derived from this order. Using these sequence numbers as branch priorities, a DCC-based replication algorithm in the metadata servers can ensure that always the latest update to the metadata is effective.

Low change rate. For DCC, the odds of having to cut a branch depends directly on the communication latency between replicas and the probability of conflicting change attempts during one round-trip time. A low change rate should be given for

example for many user-facing web services, in which users update data manually. Their updates will usually reach only one replica and are separated by relatively long idle intervals that are determined by the interaction speed of the user with the web site.

In this case, concurrent updates on multiple replicas are only imaginable in case of failures and retries. Thus, while concurrent changes to the replicated data can not be ruled out completely, they should be extremely rare and are therefore probably not worth paying for the cost of coordination of each operation. The resulting loss of updates should be negligible given the large amount of interactions with the web site, and bearable for non-critical data such as social profiles.

Non-atomic fail-over. Master/slave replication [67] systems usually rely on an external mechanism that selects a master and redirects clients to a new master on failure. This fail-over can be made atomic in order to ensure that there is always only one master in the system [65]. However, when those fault-tolerant leader election algorithms are out of scope, developers often rely on simple heartbeat-message driven mechanisms that take over the IP address of the master or they rely on fail-over logic of layer 7 switches (load balancers).

These simpler fail-over mechanisms are usually not atomic, i.e. it can happen that for a short time two replicas exist that receive requests from clients that change the same data. This is not unlikely, because clients might retry their write request. In this case the DCC algorithm can be applied to order these requests in a consistent manner. In addition, when the fail-over mechanism is able to count master selection rounds, the resulting order can be made part of the write operation priority. This ensures that later requests always supersede older requests.

Distribution-aware applications. Replication systems that follow the ROWA [67] approach are based on a replicated state machine achieve replica consistency by coordinating access to data between replicas. Through this coordination, these replications system avoid concurrency of changes in the first place and thereby provide applications with total order.

With this mechanism, applications are able to access replicated data in a transparent manner that is similar to a single copy of the data. The downside of this approach is two-fold. First it comes with the cost of communication for ensuring the total order, including inevitable round-trip delays. Further, the transparency of the approach requires the replication system hides all failures once the application has been signaled success.

Using DCC, a replication system can implement a weaker version of change coordination. The application is made aware of the result of its operation. As soon as an operation passes the all-know cut, no further concurrency is possible and the correct execution of the change can be acknowledged to the application. In case of concurrency, if the application's branch had to be cut, the application can be signaled and retry its modifications to the data.

6.4.3 Specification

The Distributed Consistent Cutting (DCC) algorithm maintains eventual consistency for change operations as we have defined them in section 4.2:

- When operations are **causally dependent**, the algorithm **maintains their causal order**.
- When operations are **concurrent**, the algorithm eventually select one **effective branch** and **guarantees a total causal order** for its changes. The selection of a branch is eventually consistent across all replicas. All other branches are dropped and their changes are suppressed.
- In addition, we require that a replica may only issue an operation if it is not aware of any concurrent attempts to change a value[1].

DCC as an Event Handler

From the perspective of the replication system, DCC has to be an **event handler** that handles operations as it receives them and appends them to the log. Because our dissemination mechanisms guarantee at least FIFO delivery (or even causal delivery), we only need to **define the logical log order between the latest operations from each host**. Any earlier operation from a host is received before its causal successor and therefore will be overwritten by it.

Further, DCC must **solely rely on the ordering and re-ordering primitives**, which define the logical order of operations in the log (see Sec. 5.3.2). The logical log order is a total order whose maximum in the context of DCC is the **effective operation** that determines the current state of the replicated data. In case of concurrency, this effective operation is the latest operation on the effective branch and overwrites the effect of all concurrent operations and therefore the branches that they have created implicitly are cut.

Consequently, the DCC algorithm needs to find the effective operation among the latest operations and enforce it with the logical log order. Thereby it defines a strict **total order relation $<_{dcc}$ between the latest operations** from each host.

Consistent Branch Selection

When selecting a branch over other branches, DCC shall prefer a branch over other branches according to an application-defined measure. In order to allow an application to have control over this preference, we allow the application to provide a **concurrency order $\leq_c$** that defines a preference measure.

A concurrency order $\leq_c$ needs to have the following properties:

[1] We have already introduced a similar restriction for DCB, for which applications may not further change the default branch if conflicting updates and therefore dependent branches are known. This restriction will be important later to be able to argue for eventual consistency.

1. In order to be practical, $\leq_c$ needs to define a *non-strict* total order between change operations. As a non-strict total order, $\leq_c$ implies a definition of equivalence, $=_c$.

2. Operations from different hosts need to be strictly ordered:
   $process(o_1) \neq process(o_2) \Rightarrow o_1 \neq_c o_2$.
   This can be achieved by adding a small-grained host specific offset.

The totality of $\leq_c$ ensures that the order may be defined for any pair of operations. As a non-strict total order, any $\leq_c$ definition can be mapped to natural numbers. Therefore we can think of defining $\leq_c$ as assigning "priorities" to operations.

The definition of the concurrency order $\leq_c$ leaves some freedom in finding an ordering criterion that is suitable for an application. It is possible to imagine definitions for $\leq_c$ that rely on information from any operation along a branch. The $\leq_c$ can even change with the reception of new operations as long as it is eventually consistent.

The following definitions of a concurrency order are conceivable:

- We can use a process' local clock to associate a timestamp with each operation, or make it part of the event itself. Then operations that were issued earlier (in terms relative of the local clocks) can be made having a higher priority.
- We can use random numbers to ensure a certain fairness when selecting effective operation.
- We can use a total order from an external source, such as an external sequencer.
- We can use priorities that are generated by a master selection mechanism.
- We can rely on introspection of an operation in order to prioritize certain operations over others. For example, setting a flag could be prioritized over erasing it, allowing boolean conjunctions.

In order to better understand the following sections, we can thinke of the concurrency order as being derived from a static order among hosts. In this exemplary definition of $\leq_c$ the source of the operation would determine its priority in case of concurrency. Using static priorities *prio* assigned to the replica hosts, this results in the following definition of the concurrency order:

$$o_1 \leq_c o_2 :\Leftrightarrow prio(process(o_1)) < prio(process(o_2))$$

Specification of $<_{dcc}$: Causally-Consistent Branch Selection

In order to meet DCC's specification, we need to enforce the application-defined concurrency order in a way that is consistent with causality. Otherwise we would not guarantee that DCC maintains causal order for non-concurrent operations. This

is not trivial, because the concurrency relation is not transitive. Therefore applying the application-defined concurrency order directly to concurrent operations violates causal ordering.

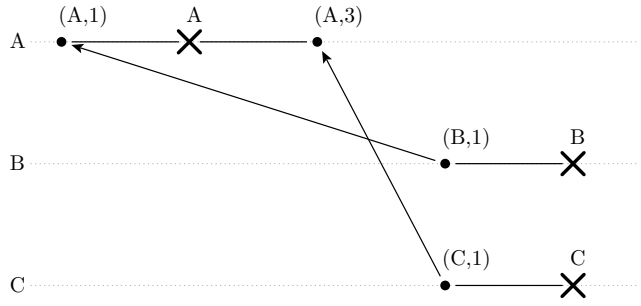
Consider the following example that uses a static order among hosts for the definition of $\leq_c$:

$$C \leq_c B \leq_c A \text{ (concurrency order)}$$

Further assume that:

$$A \longleftarrow C \text{ and } A \parallel B \parallel C$$

As a causality graph:



Therefore we get the following inconsistent pair-wise orders:

$$A \parallel B, o_b \leq_c o_a \Rightarrow o_b <_{dcc} o_a$$

$$B \parallel C, o_c \leq_c o_b \Rightarrow o_c <_{dcc} o_b$$

but

$$A \longleftarrow C \Rightarrow o_a <_{dcc} o_c$$

We see that we can not directly apply the concurrency order to the latest operations from each host. DCC needs to define a strict total order between the latest operations from all replicas. Directly applying the concurrency order to the latest concurrent operations results in an intransitive relation and therefore not in an order.

In order to yield a strict total order between the latest concurrent operations, we resort to the concept of branching. Because of our restriction that conflicted keys may not be changed, we know that each operation has only one causal predecessor, and therefore concurrent operations form branches. For an operation, we can define the function $latest(o)$ that yields the **endpoint** (latest write) of the branch it lies on. With its help, we can define the order that DCC establishes between concurrent operations:

$$o_1 <_{dcc} o_2 :\Leftrightarrow \begin{cases} o_1 \not\parallel o_2 : & o_1 \longleftarrow o_2 \\ o_1 \parallel o_2 : & latest(o_1) \leq_c latest(o_2) \end{cases}$$

The result is a strict total order between the latest operations from all hosts. The maximum from all received operations is the **effective write**:

$$o_e = max_{<_{dcc}}(o), o \in L$$

$<_{dcc}$ is a strict order because the concurrency of two operations implies that they lie on different branches:

$$o_1 \parallel o_2 \Rightarrow latest(o_1) \neq latest(o_1)$$

Therefore their latest operations come from different hosts, otherwise they would not be on different branches. From the definition of $\leq_c$, it follows that the that these operations must be strictly ordered:

$$process(latest(o_1)) \neq process(latest(o_1)) \Rightarrow latest(o_1) \neq_c latest(o_1)$$

Therefore, $<_{dcc}$ is consistent with causality.

6.4.4 Correctness: Eventual Consistency of DCC

We have just seen how the specification of DCC can be implemented using the ordering and re-ordering primitives of the log and how this implementation maintains causal order. We now want to argue for its correctness. Just as DCB, DCC is a reactive algorithm and will always make progress as long as all replicas are available. We will discuss how to handle absence of replicas in section 6.4.8. We now want to **argue for the safety of DCC**, which means that the execution of DCC results in an order $<_{dcc}$ that is eventually consistent across all replicas.

In order to properly define $<_{dcc}$ for an operation o, we need to define the operation's relation to all its concurrent operations $\bar{o}$. Because the concurrency relation $\parallel$ is symmetric, the set of concurrent operations $\bar{o}$ and the sets of concurrent operations $\bar{o_c}$ for any operation $o_c \in \bar{o}$ intersect in at least o. However, because concurrency is not transitive, the sets are not equal and we need to fill this gap in order to properly define $<_{dcc}$.

In Fig. 6.3, we have depicted the concurrency set of an event from B as gray areas. An event from A that is contained in the concurrency set of B has itself a concurrency set (sketched). This concurrency set has to contain the event from B due to the symmetry of the concurrency relation. In this case the concurrency sets at C overlap.

We can transitively extend the concurrency set $\bar{(o)}$ of an operation o along the concurrency relation. Let us call this set of all transitively concurrent operations of an operation o its **window of concurrency** $\Xi(o)$. Because of transitivity, the windows of concurrency of operations in the same window of concurrency are equal (if $\acute{o} \in \Xi(o)$ then $o \in \Xi(\acute{o})$) and disjunctive otherwise. Thus, containment in $\Xi(o)$ defines an equivalence relation and operations in different windows of concurrency can not influence each other.

Lemma 6.3 *The size of the window of concurrency $\Xi(o)$ of an operation o is finite.*

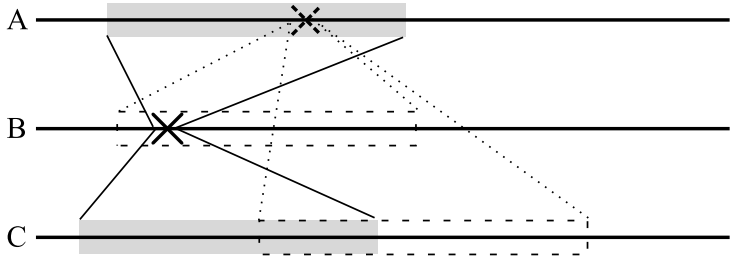


Figure 6.3: Transitive relations between concurrency sets

Proof Assume the operations in $\Xi(o)$ are ordered by the partial causal order. Then there is always a minimal element, the operation on which all operations in $\Xi(o)$ are dependent. In case there was no concurrency before, it is the operation that all replicas knew when issuing their concurrent change attempts. Otherwise, after a period of concurrency, there will also be an operation that is causally dependent on all concurrent change attempts. This is the first operation that is issued after all replicas have received the effective operation and no further concurrent attempts to change the value can be received. $\diamondsuit$

Theorem 6.4 *The window of concurrency $\Xi(o)$ of an operation o is a subset of the all-know cut a^1.*

Proof Because all replicas know the first operation of the window of concurrency, it is before the all-know cut. Any later operation is concurrent and therefore not contained in the all-know cut. Thus, the $\Xi(o)$ is always completely contained in the all-know cut a^1. $\diamondsuit$

It follows that the change semantics of DCC result in a **cyclic behavior**:

1. During normal operation, replicas can freely change a value without coordination.

2. When there has been a concurrent attempt that has been disseminated to other replicas, the other replicas stop changing the value as soon as they learn about the conflict. The availability of the data item is then restricted to read operations (whose result can be marked accordingly). The replicas select an effective operation among the concurrent attempts as they arrive.

3. At some point, each replica knows that all other replicas are aware of concurrency and have stopped issuing changes to the particular data items. When all replicas know the same effective operation, no further concurrent attempts can occur and the value can be freely changed again and the cycle repeats.

Inside the window of concurrency, all operations after the all-know condition can potentially have concurrent operations. As we did for DCB, we bind these operations together to *branches*. Because the window of concurrency is finite, there will be a last operation on each branch that modified the value within the window. The latest operation with the highest concurrency order $\leq_c$ will define the final value consistently on all replicas.

6.4.5 Distributed Consistent Cutting – General Version

We will now develop a general blueprint for the operation of the Distributed Consistent Cutting algorithm. We will adapt this blueprint in the next sections to implementations of DCC for mobile and server environments.

The general operation of DCC is the one of an online algorithm. It receives a remote event e_r and decides its effect on the local data set. This decision is based on the contents of the local log, i.e. the history of received and locally generated events. It controls the effect of received operations on the local data set by using the ordering and re-ordering mechanisms.

When handling a received operation, DCC finds a data set from the log that contains the writes that it has already received and handled. For implementing DCC, we are only interested in those operations that lie behind the I-know-that-all-know cut a^1 in the log, which contains the window of concurrency. Operations before that cut can not be concurrent to any operations we receive in the future. This also means that DCC requires only to keep operations after the all-know cut.

When DCC handles a received operation, it needs establish its relation to any existing concurrent branches. For branches, we can tell concurrent from non-concurrent endpoints (see DCB). Because we only regard operations within the window of concurrency, operations have at most one causal predecessor. Thus at most one non-concurrent endpoint exists.

The received operation e_r can relate in three ways with the branches, resulting in three cases to be differentiated:

1. It can extend an existing branch. Then it will be a causal successor to the non-concurrent endpoint.

2. It can be inside an existing branch. This can happen because causal dissemination does not guarantee that all operations of a branch are received in order (this happens when the branch changes between replicas).

3. It can create a new branch.

This leads us directly to a **general version of the DCC algorithm** (see algorithm 5). We can tell three cases:

1. The new effective operation is the remote operation. That means that it can be directly executed (line 6–7).

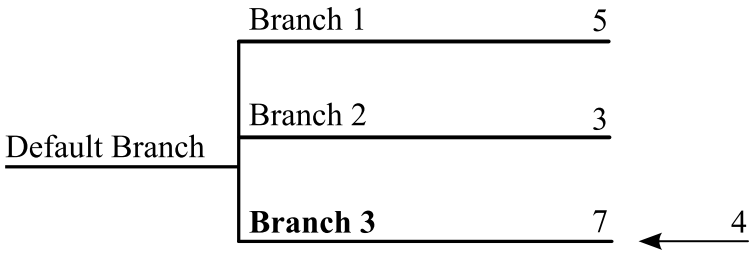


Figure 6.4: Assume branch 3 is effective because its latest write has the highest priority 7. A new operation on that branch might change its priority to 4, leading to the new effective branch 1.

2. The new effective operation is the old effective operation. This means that the remote operation o_r needs to be suppressed when appended to the log (line 8–9).

3. The effective operation changes to another operation from the log. Then this operations need6s to be re-ordered by re-executing it with the redo mechanism (line 10–11).

Algorithm 5: handleRemoteOperation(o_r) – General Blueprint

```
1  begin
2      compute branches and their endpoints E;
3      determine current effective operation o_e = max_{≤_c}(E);
4      apply e_r to its branch;
5      recompute branches and new effective operation õ_e ;
6      if õ_e = o_r then                    // o_r is the new effective operation
7          execute o_r;
8      else if õ_e = o_e then                  // effective operation unchanged
9          suppress o_r;
10     else if õ_e ≠ o_e then                   // other operation is effective
11         suppress o_r;
12         redo õ_e;
13     end
14 end
```

6.4.6 Distributed Consistent Cutting – Mobile Replication

We will now adapt the general DCC algorithm for mobile systems that use **reconciliation via causal gossip** for communication. With causal gossip, a replica receives a set of new events containing multiple operations and handles each event individually. Because reconciliation is a comparatively rare event, we can generate

any necessary state that is required for the execution of DCC on demand and retrieve all necessary information from the log.

The amount of data that we have to retrieve from the log is bound. Recall that we are only interested in events that lie in the window of concurrency. As it is fully contained in the all-know bounds, we can stop retrieving events when reaching the all-know cut a^1(algorithm 6, line 2).

Because all relevant operations from the log are available when importing a remote event, DCC can rely on them as context when determining the DCC order $<_{dcc}$. This allows us to introspect operations for a semantic definition of the $\geq_c$ order. Furthermore, because the algorithm can re-determine the $<_{dcc}$ for each imported operation, we can even allow definitions of $\geq_c$ that become only eventually consistent as all concurrent operations have been received.

Algorithm 6: handleRemoteOperation(o_r) - Causal Gossip Version

```
   Data: L the local log, C causality graph from events e > a^1 from L
 1 begin
 2     e_n, E_c = computeEndpoints(o_r, D)   // Algorithm 1;
 3     o_e = max_{≤_c}(E_c ∪ e_n)   // determine current effective operation;
 4     if o_r ⟶ o_e then                    // o_r on current effective branch
 5         o_a = max_{≤_c}(E_c)   // alternative eff. operation;
 6         if o_a ≥_c o_r then          // o_r lowers priority of its own branch
 7             suppress o_r;
 8             redo o_a   // other branch becomes effective;
 9         else
10             execute o_r   // eff. branch of o_r remains effective;
11         end
12     else
13         if o_r ≥_c o_e then                    // o_r is new effective branch
14             execute o_r     // o_r becomes new eff. operation;
15         else
16             suppress o_r  // current eff. operation remains effective;
17         end
18     end
19 end
```

The first step of DCC is the calculation of concurrent and non-concurrent endpoints using the same basic algorithms as DCB uses (algorithm 6, lines 2-3). Then we determine the current effective operation from the set of all endpoints. If the remote operations extends the current effective branch, we have to check whether another branch becomes effective, because o_r lowers the priority of its own branch too much (lines 7-8) or the current effective branch stays effective (line 10).

If o_r is not extending the current effective branch, it will become an endpoint. Depending on whether its priority exceeds the priority of the current effective branch

(line 13), o_r becomes the new effective branch (line 14) or the current effective branch stays effective (line 15).

6.4.7 Distributed Consistent Cutting – Online Replication

The DCC algorithm as presented in the previous section will find concurrent operations by retrieving information from the event log and inferring the total order comparing dependency vectors. This is feasible in mobile replication systems where the operation rate is relatively low and each event contains many operations that share causal information. In an online replication system, we usually face much higher operation rates and therefore it is necessary to **minimize the work**, which is necessary to import a received operation.

Instead of scanning the log to infer how we have to handle the received operation, we will keep state that summarizes what we would have found in the log and act on it. This state has to fulfill two purposes:

1. for a received operation e_r we need to be able to find out whether there is any concurrent operation present in the local log,

2. in case of concurrency, we need to be able to determine the effective operation.

For these two purposes, we **keep a matrix that summarizes the latest concurrent operations** in the log of a certain key. This matrix K_{ij}^k per key k records the dependency vector from the latest received operations that come from process i as a column vector:

Definition 6.7 *K_{ij}^k for key k is a matrix whose rows are composed of the dependency vectors of the latest received operations of the replica A, B, C, ...:*

$$K_{ij}^k = \begin{pmatrix} \vdots & \vdots & \vdots \\ dep_{latest}^A & dep_{latest}^B & dep_{latest}^C \\ \vdots & \vdots & \vdots \end{pmatrix},$$

The column vectors of K_{ij}^k can be **grouped to branches**. If two column vectors are causally dependent, they belong to the same branch. Let us call the partitioning of K_{ij}^k along this equivalence $\bar{B^k}$. We will use this matrix now to determine the effective operation after the reception of a new operation o_r in three steps:

1. We compute the set of branches,

2. determine the branch with the effective operation,

3. and map the found order to the physical log order.

Computing the set of branches can be done solely with the information in K_{ij}^k. Any received operation o_r will have a causal relationship with one of the branches in

$\bar{B^k}$, or define a new branch. Because the direct send allows causally related operations from different hosts to arrive in any order, received operations do not necessarily extend a branch, but can also lie on one of the established branches. Furthermore, we need to detect false concurrency and assign the operations to their correct branch.

The effective operation is determined by applying the concurrency order $\geq_c$. Some definitions of the concurrency order $\geq_c$ require keeping further information for determining $\geq_c$ between to operations. A common use case will be including some kind of priority measure with each operation and using that directly as the definition of $\geq_c$. In this case, we also need to keep a **priority vector** p^k with each K_{ij}^k matrix. This vector will store the priority information that came with the latest operation from a particular node.

The received operation can have an effect of the order of operations in multiple ways. The received operation can become the effective operation or be entirely without effect on the current data set in which case it must be suppressed. However, because causal ordering has precedence over concurrency order $\geq_c$, the received operations may also cause the branch with the effective operation to be cut in favor of another branch, resulting a re-ordering of operations. Together, this leads to the following differentiation of cases:

1. The operation o_r does neither extend an existing branch nor creates a new branch. Then it needs to be ordered before the other operations in effective log order by using suppression.

2. The operation o_r creates a new branch. Depending on its position in the concurrency order $\geq_c$, it either needs to be suppressed or becomes the effective operations.

3. The operation o_r extends an existing branch. The operation may become the effective operation, the effective operation may be unchanged, or another operation becomes the effective operation.

Algorithm 7 shows the complete DCC algorithm for online replication (without operations on the p^k vector). First the branches are computed as the branch matrix $\bar{B^k}$ (lines 2–9). Then we handle the case if a causal successor of o_r has overtaken o_r (FIFO delivery) and already reached the node. In this case o_r needs to be suppressed (lines 10–11). If o_r creates a new branch, o_r becomes the new effective branch if it dominates all existing branches, otherwise it is suppressed (lines 12–17). If o_r extends an existing branch, it is executed if it dominates all branches (lines 19–21). Otherwise it is suppressed and another branch is re-instantiated as the effective branch if necessary. Finally the dependency vector of o_r is absorbed by the K_{ij}^k matrix.

Algorithm 7: *handleRemoteOperation*(o_r, k, p) on key k with priority p

Data: K_{ij}^kmatrix for key k
Result: flag to indicates whether to suppress the operation

```
1  begin
2      B̄^k = ∅ ;
3      forall columns j do
4          if ∃b ∈ B̄^k : K_j^k ⟶ b then
5              replace b in B̄^k with K_j^k;
6          else if ∀b ∈ B̄^k : K_j^k ∥ b then
7              B̄^k = B̄^k + K_j^k;
8          end
9      end
10     if ∃b ∈ B̄^k : b ⟶ dep(o_r) then                    // o_r on existing branch
11         suppress o_r;
12     else if ∀b ∈ B̄^k : b ∥ dep(o_r) then                // o_r new branch
13         if ∀b ∈ B̄^k : o_r ≥_c b then                    // o_r dominates
14             execute o_r;
15         else
16             suppress o_r;
17         end
18     else if ∃b_B ∈ B̄^k : b_B ⟵ dep(o_r) then            // o_r extends branch b_B
19         if ∀b ∈ B̄^k : o_r ≥_c b then                    // o_r dominates
20             execute o_r;
21         else
22             suppress o_r;
23             if ∀b ∈ B̄^k : b_B ≥_c b then                // b_B was effective
24                 redo b : max_≥c(b);
25             end
26         end
27     end
28     K^k_process(o_r) = dep(o_r);
29 end
```

Example Assume the latest received operations for a key k were:

$$dep(o_A) = \begin{pmatrix} 5 \\ 2 \\ 1 \end{pmatrix} priority(o_A) = 8 \text{ from process } A,$$

$$dep(o_B) = \begin{pmatrix} 3 \\ 9 \\ 2 \end{pmatrix} priority(o_B) = 5 \text{ from process } B,$$

$$dep(o_C) = \begin{pmatrix} 8 \\ 3 \\ 7 \end{pmatrix} priority(o_C) = 7 \text{ from process } C.$$

These dependency vectors along with their priorities result in the matrix

$$K_{ij}^k = \begin{pmatrix} 5 & 3 & 8 \\ 2 & 9 & 3 \\ 1 & 2 & 7 \end{pmatrix} p^{k\ T} = \begin{pmatrix} 8 & 5 & 7 \end{pmatrix}.$$

from which the following matrix of branches can be computed:

$$\bar{B^k}\ |p^k = \begin{pmatrix} 3 & 8 \\ 9 & 3 \\ 2 & 7 \end{pmatrix} p^{k\ T} = \begin{pmatrix} 5 & 7 \end{pmatrix}.$$

Therefore the effective operation is on branch 2 and has the priority 7.
The reception of an operation o_r with

$$dep(o_r) = \begin{pmatrix} 9 \\ 11 \\ 7 \end{pmatrix} priority(o_r) = 8$$

would extend the second branch and increase its priority. Therefore it would become the effective operation and be executed on the local data replica.
The reception of an operation o_r with

$$dep(o_r) = \begin{pmatrix} 9 \\ 11 \\ 7 \end{pmatrix} priority(o_r) = 4$$

would extend the second branch and decrease its priority below the one of branch 1. Therefore branch 1 would become the effective branch and its operation would re-executed on the local data replica (redo mechanism). o_r would be logged as suppressed.

And as the last case, the reception of an operation o_r with

$$dep(o_r) = \begin{pmatrix} 3 \\ 10 \\ 3 \end{pmatrix} priority(o_r) = 6$$

would extend the first branch. Due to its lower priority, however, branch 2 would remain effective and o_r would be logged as suppressed.

Garbage Collecting K_{ij}^k Matrices. When some data has not been changed for a few message exchanges, the latest change operation will eventually fall behind the all-know cut $a^1$and there is no further possibility for concurrency. In this case, we can delete the K_{ij}^k matrix along with any priority and redo information. We treat a non-existing K_{ij}^k matrix as the zero matrix. As soon as we receive the first change operation, the matrix is in existence again and will track any concurrency and allow establishing order among concurrent operations.

We can detect that a particular matrix for a key is obsolete in this sense by maintaining an overall matrix K_{ij}^* , which records the dependency vectors of operations on any key. If this overall matrix dominates a key's matrix K_{ij}^k, K_{ij}^k along with its priority vector p^k can be deleted.

Algorithm 8: *garbageCollectKeyMatrices*(o_r,k) after receiving o_r on key k

```
   Data: K_ij^k matrices, K_ij^*
1  begin
2      i = process(o_r);
3      K_i^k = dep(e);
4      K_i^* = dep(e);
5      if K_ij^k < K_ij^* then
6          delete K_ij^k and p^k ;
7      end
8      K^*_{process(o_r),k} = dep(o_r);
9  end
```

6.4.8 Failure Detection and Recovery

Because DCC is supposed to be run as part of a real replication system, it has to be able to handle failures of hosts and networks. In particular, we need to be able to handle network partitions and crashing and recovering hosts in a way that does not compromise the consistency of replicas. This problem has two aspects: how can a process recover from a crash, and how can the rest of the system handle a non-responsive process.

Recovering from crashes

When a process has crashed and recovers, it has to update its local replica to the current state so that it can continue participating in the replication system. It can do so in two steps:

1. Request all missing events from one or more hosts by sending them its summary vector along with a request to send them all events that have happened since them. Because all hosts store events along with full causality information, and because DCC handles ordering between hosts, any subset of processes can fulfill this request in parallel.

2. Start participating in the direct send event broadcast, but suspend local delivery until all missing events have been received and delivered locally (obey FIFO order).

In order to correctly embed the requested operations in its state, the node needs to run the DCC algorithm. To rebuild the necessary state, the node can replay its local log into the DCC algorithm without actually executing the operations. When it replays operations starting from the all-know cut a^1, all context information will be in place to correctly import any received operation.

After these two steps, the replica has caught up and is a passive member of the replication system and has current data. It now can start generating events and disseminate them. If events of the time before the crash have not been delivered by all replicas, they will detect a gap in the sequence of events and re-request them.

Replica Consistency

Because DCC is an event-driven algorithm, crashes and network partitions are not a problem per se. DCC will not have to wait for any messages from remote processes. However, these failures can introduce long periods of concurrency. If a process has crashed and re-sends events during recovery, these events can be concurrent to events from other processes that have been sent hours later and influence their outcome. Also, if the replication system has been split into two or more isolated partitions by a network problem, these partitions are not per se hindered in continuing their work. However, this mode of operation will also result in a long period of concurrency that affects operation over a long time.

Our goal is to restrict these periods of concurrency in a way that controls how long operations can be affected. For this purpose, we can integrate DCC with a group membership algorithm [53]. The group membership algorithm decides which processes are alive and form the current replication system. Furthermore, it can provide a total order between these views of the replication system. By embedding the current view number of a process $view(p)$ in its change events, we can control the length of the window of concurrency. We define the view number $view(e)$ of a change event e as the maximum view number at its time of creation $view(p)$ and of all its causal predecessors.

We also define the view number of a window of concurrency as the view number of the first change event. As new concurrent events are received and branches are identified, the view number can change to earlier events, making it monotonically decreasing.

The view number of a change event is influencing the operation of DCC in two ways:

1. The view number of a change event is part of an event's priority and determines the $\leq_c$ order when it differs between events.

2. The view numbers determine the length of the window of concurrency. Because an event with a higher view number always takes precedence, any concurrent events with a lower priority can be disregarded and data can be modified again.

In effect, this redefines the concurrency relation between two events. Two events are concurrent when their dependency vectors indicate so and they belong to the same view. Note that this definition is an extension of causal order, because the view number of an event is always equal or larger than the one of its causal predecessors.

Chapter 7

Evaluation

The LogDB Database system is a replicated database that supports offline replication with reconciliation of mobile devices and online replication of server databases. LogDB was implemented as a proof-of-concept for our log-centric approach to replication. It acts as a container for our replication algorithms and mechanisms and is an experiment that investigates how well the simplicity of the log-centric approach translates to a simple design and implementation of a database system. LogDB is not a full database management system and not optimized for performance: the database lacks certain components such as a query optimizer, and cross-table functionality like relational integrity or joins[1]. For that reason, we restricted ourselves to investigate chosen aspects that are unique to our approach and refrained from running standard database benchmark suites. The resulting measurements shall give an estimate on how our algorithms behave in practice and allow a comparison with alternative approaches.

Like relational databases, LogDB provides tables with typed columns that are defined by table schemata. However, LogDB does not implement relations between tables and uses a fixed primary key, the row identifier. In order to support the replication algorithms of Chapter 6, the database can keep multiple branches of a row. LogDB is also able to track reconciliation operations and event dependencies and can compute all-know conditions in an efficient manner.

LogDB employs a **log-only architecture** as described in section [62]. The persistent log and the B-trees of the indices are mapped into memory. Changes to the data are persisted by appending them to the log, flushing the append to disk, and updating the indices. The state of indices is well-defined by the event identifier of the latest operation in the log. However, in order to simplify the implementation, no checkpointing mechanism has been implemented and indices are only updated on disk by a graceful shutdown of the system. When the system crashes, indices are rebuilt by replaying the log and executing the operations on them. Our garbage collection algorithm ensures that operations that do not contribute to the current

[1] We believe that in general, no aspect of our design should prohibit a high-performance implementation in a full database management system. Even more, the choice of a persistent log as the core data structure should support such an implementation.

state of the database are removed from the log and thus the log replay only has a limited amount of operations whose effect is overwritten by operations later in the log.

LogDB is designed as an **embedded database** for applications that need replicated structured storage. In contrast to DBMSs that are accessed via socket connections, applications use LogDB in a blocking synchronous manner, i.e. either per function call or by sending command events to it and waiting for the results. LogDB assumes that concurrency is handled within the application and it does not support coordination of concurrent operations within a replica. In effect, LogDB only has to handle local operations, which it executes atomically. This simplifies database design because there is no need to implement elaborate locking or commit/rollback mechanisms in the database. Specifically, no locking in indices is necessary and operation atomicity can be ensured without a redo/undo log by preparing operations before executing them so that no rollback is ever necessary.

7.1 Distributed Consistent Branching

In our first experiment, we **compare** the resource usage of our Distributed Consistent Branching (DCB) algorithm with a **version vector-based alternative**. While DCB is an algorithm for a log-based replication system, its semantics and application domains are comparable to those of state-centric algorithms (see Sec. 2.6). The target system of this algorithm is mobile and characterized by a moderate number of nodes, relatively low write frequencies, a large number of small data items, and a general shortage of communication and storage capacity.

State-centric replication algorithms normally use version vectors to track dependency between versions on different hosts. The version vectors allow them to distinguish creations from deletions, for example. Version vectors represent a constant overhead for every data item. The resource usage of our log-based approach, in contrast, is heavily dependent on the dynamic parameters, and it is better to assess this overhead with an experiment.

The **version vector-based algorithm** is derived from other systems that have been proposed for mobile environments (such as Rumor [55], Ficus [111], or Bengal [36]). In these implementations, each key branch is accompanied by a version vector. The node that initiates the reconciliation process requests all version vectors from another (source) node, compares the vectors key-wise to its local copies, and formulates requests for updated or conflicting data from the component differences. When conflicting changes to a key are detected, the key is branched to reflect the conflicting versions of the associated data.

We evaluate the two algorithms in a simulated mobile environment using our own discrete event simulator. In the simulation, both algorithms process the same write and reconciliation operations but work on isolated data stores and communication paths. After each reconciliation operation is processed, the data sets of the version vector algorithm and our own are compared for their semantic equivalence.

We consider a number of metrics for rating the performance of the two algorithms. From an application developer's perspective, the following variables may have significant effects on both storage and communication costs and should be taken into account:

- the number of nodes in the system
- the write/reconciliation ratio and distribution
- database size, e.g. the number of data items
- data size, e.g. the average size of individual data items
- the communication/reconciliation topology

For the target systems indicated previously the most relevant metrics are write/-reconciliation ratio and database size, which we measure by calculating the size of the metadata overhead in both storage and communication. We have not investigated the effects of the reconciliation topology, but we expect that skewed topologies will exacerbate storage overhead because they allow replicas to diverge more, resulting in longer logs.

Experiment Setup

The simulator models a system of N mobile nodes. At each step a node either writes to its local data set, reconciles changes from a random (uniform distribution) source node, or merges a branched key. A node chooses an operation based upon the reconciliation probability p, taken from a uniform distribution: reconcile with probability p, merge with probability p, or write to a randomly chose key with probability $1-2p$.

Our first simulation investigated the effects of varying the number of data items. We ran our simulator with four nodes and a fixed reconciliation/write ratio of 1:9, starting from a state in which all nodes have a reconciled data set consisting of between 100 and 2000 keys.

Our second simulation measured the influence of reconciliation probability on storage and communication overhead. A higher reconciliation probability p reduces the average number of writes (occurring with probability $1-2p$) between two reconciliations and thus decreases divergence in the system. We measured the resulting overhead in a system of four nodes with 1000 keys each. The reconciliation and merge probabilities varied between 0.01 and 0.4.

In the third simulation we increased the number of nodes in a system where each node had 200 keys and reconciled and merged each with a probability of 0.1.

For each run we measured the average metadata storage overhead per node and of all nodes together, the total communication overhead volume of the system, and the average communication overhead per reconciliation operation during the last quarter of the simulation time (to mitigate any ramp-up effects prior to reaching the

branching-merging equilibrium). Each simulation ran ten times for 1000 timesteps and the results were averaged across runs.

Results

As expected, the storage overhead of the version vector implementation (Fig. 7.1, dashed line, log scale) increased in direct proportion to the number of keys. The log-based approach, in contrast, does not require any per-key metadata, so increasing the number of keys has no significant effect (Fig. 7.1, solid line). Because the version vector implementation has to transfer version vectors for all data items at each reconciliation, its total communication overhead depends heavily on the database size (Fig. 7.2, dashed, log scale).

The log algorithm benefits from diminishing write frequency (Fig. 7.3, solid line, log scale), because smaller divergence between replicas lets the algorithm purge the log more aggressively, so fewer write operations have to be stored. In contrast, the version vector algorithm has a larger, constant storage overhead (Fig. 7.3, dashed line, log scale).

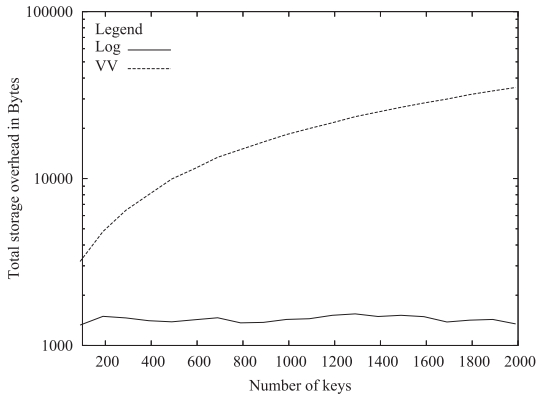


Figure 7.1: Storage overhead for increasing data set size for a log-based and version vector-based implementation.

Increasing the number of nodes has a noticeable effect on the per-node storage overhead of the log-based system (Fig. 7.4, solid line). Adding more nodes to a system when the reconciliation frequency is constant results in a larger divergence of the replicas, which in turn makes the log grow longer. There is also a slight increase in overhead in the version vector system, which is due to the higher conflict probability inherent in a more diverged system (Fig. 7.4, dashed line).

Discussion

We observe that the metadata storage overhead of our log-based algorithm is dependent on the relative divergence of the replicas, which is caused by infrequent reconciliations or frequent changes, but is agnostic to changes in the size of the data

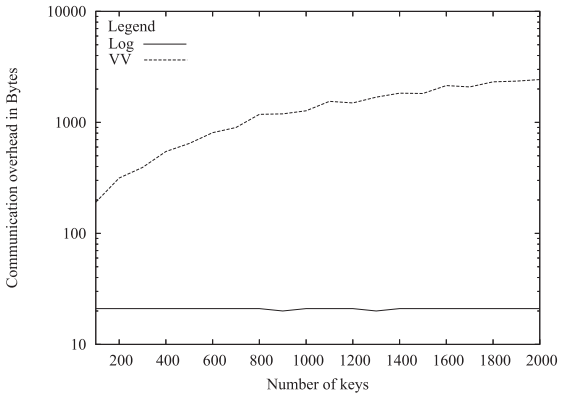


Figure 7.2: Communication overhead for increasing data set size for a log-based and version vector-based implementation.

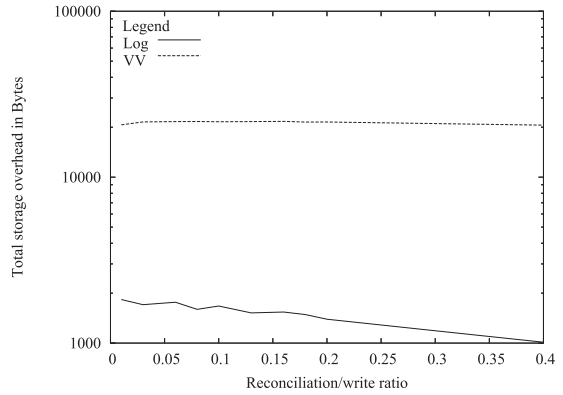


Figure 7.3: Storage overhead for increasing write frequency a log-based and version vector-based implementation.

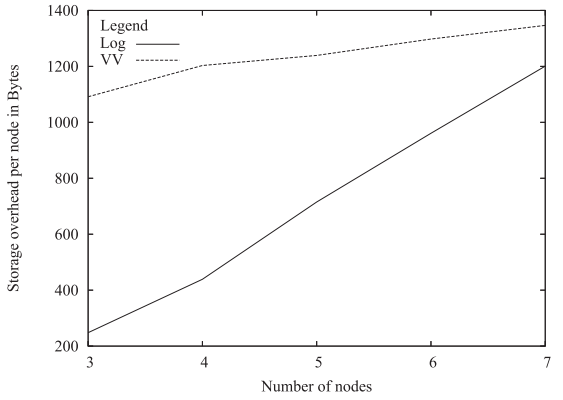


Figure 7.4: Storage overhead for increasing replication system size for a log-based and version vector-based implementation.

set. The communication overhead our algorithm is directly proportional to write frequency in the system, since the algorithm only transfers changed data. Contrast the version vector-based system, which has a communication and storage overhead that is proportional to the number of data items and is agnostic of change rates.

We conclude that a personal database system with low write frequencies and a large number of small records can significantly benefit from the application of our log-based algorithm. When data records are larger and the size of records dominates the measured savings in metadata storage and communication overhead, a version vector-based algorithm should be considered due to its simpler implementation.

7.2 Distributed Consistent Cutting

With the following experiment, we assess the resource usage and performance of the Distributed Consistent Cutting algorithm under varying operating conditions. Recall that in order to avoid having to retrieve data from the log, DCC caches causality information in dynamically created **per-key matrices**. These matrices are an artifact that can be considered overhead of the algorithm. As they are only created and maintained as long as concurrency is possible, they can also be used as an indicator how much potential concurrency is in the system. In our experiments, we will quantify the overhead they represent by measuring the number of concurrently existing matrices. We will also track the individual lifetime of the matrices as an indicator how long concurrency is possible.

DCC will detect any concurrent changes to the data and select one branch of changes as the effective branch. This selection process is implemented as ordering and re-ordering of operations to achieve the correct effective log order. In our experiments, we monitor the number of those ordered operations and the total number of operations that a node receives. The ratio of these two, the **reorder-ratio** indicates how much actual concurrency has been observed.

In summary, the **dependent variables** in our experiments are:

1. The average lifetime of a matrix (a measure for the *potential concurrency*).

2. The maximum number of matrices that co-existed during the experiment run (a measure for the *actual resource requirements* of DCC).

3. The ratio of reordered operations (ordered operations/total operations, a measure for the *actual concurrency*).

We observe these variables while **scaling the replication system along several dimensions**. For system scalability, the number of replicas is probably the most interesting. We also adjust the link latency to simulate replicas with longer distance and slow/congested replicas. The third variable we control is the size of the data set that is being changed ("hot" data set). Because our replication algorithms

are agnostic of the size of the data set, we do not investigate it as a scalability measure, but as a factor that influences concurrency. A larger hot data set decreases the probability that any one data item is changed.

To summarize, our **independent variables** are:

1. The link latency between replicas.
2. The size of the hot data set.
3. The number of replicas in the replication system.

7.2.1 General Setup

We conducted these experiments using our log-only LogDB database. While the core database is written in C++, the operation handling and DCC logic is Python code. Our experiments ran on a **cluster of machines**, each of which hosted one replica. The cluster connects the machines with a high-speed, low-latency (< 1ms) network. In order to simulate higher network latencies, we used a capability of the Linux kernel called *netem* that allows us to introduce queuing delays in controlled manner with a certain latency and jitter.

In this experiment the replicas executed operations generated by an embedded **load generator**. The load generator submitted batches of change operations to the database in static intervals (an open loop benchmark with closed loop batches, according to the taxonomy of [105]). The submitted requests were changing a fixed number of data items (the "hot" data set) uniformly distributed. The load generator also kept track of the completion status of the issued operations and allowed us to detect overload of the database. It signaled overload when the previous batch of operations is not processed to completion at the time when a new batch is to be submitted.

7.2.2 Calibration

Our first experiment investigates an artifact of our acknowledgment mechanism that relies on causality information. Because events implicitly piggyback reception acknowledgments for other events with their dependency vectors, the time-to-acknowledgment as observed by the sender of an event is dependent on the operation rate of the receiver. Therefore the round-trip time not only includes the latency of the link, but also a derivative of the operation rate and queuing times.

This mainly affects the detection and deletion of obsolete per-key matrices. Recall that per-key matrices are generated when an operation for a key is received and kept until we receive causal dependencies on this operation from all replicas. For this to happen, the respective operation has to be received by the third node, handled there, a new operation with a causal dependency on it has to be issued and sent eventually, and later received and handled by the node with the key matrix. Due to the asynchrony of weak-consistency replication, this delay has no impact on system performance, but it does influence the resource usage of the algorithm.

Experiment Setup We ran a setup of three nodes with 1 ms round-trip time and varied the operation rate of the load generator while measuring the effect on the variables matrix lifetime, reorder probability and number of matrices.

Results Fig. 7.5 - Fig. 7.7 show the results of the measurement both as a table and as a graph. The 95% confidence interval has been plotted as error bars.

The measurements show (Fig. 7.5) a clear influence of the operation rate on the lifetime of matrices. At an operation rate of 30 ops/s (inter-batch spacing 100 ms) the mean lifetime of an matrix was 111 ms, at an operation rate 150 ops/s (inter-batch spacing: 4.7 ms) it was at 21 ms. With a further increasing operation rate, the mean lifetime seems to stabilize at $\leq$ 15 ms.

The influence of the operation rate on the number of concurrently existing matrices (Fig. 7.6) is less distinctive. Generally, there is a slightly increasing trend for the maximum number of concurrently existing matrices, but also a large 95% confidence interval, indicating a larger variance in samples.

The third sample variable, re-order ratio (Fig. 7.5) shows a slightly increasing number which is mostly within the 95% confidence interval.

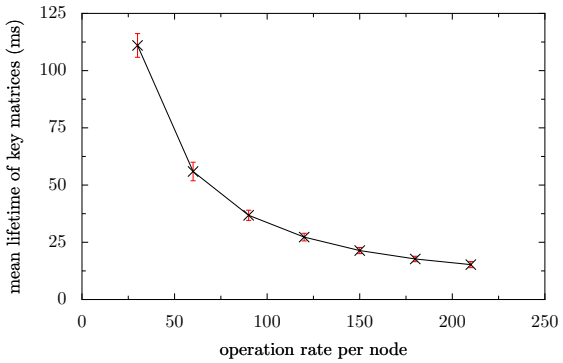
Discussion We see that operation rate has a large and distinct influence on the mean lifetime of matrices. For an operation rate of 30 ops/s we have a mean lifetime of 111 ms that needs to be accounted for. Operations are sent as a batch of 3 operations, resulting in an inter-batch spacing of 100 ms. When the rate is increased to 210 ops/s, the inter-batch spacing should be around 14 ms, which is a good approximation of the observed delays. Thus, the matrix lifetime can be explained by the time it takes the third node to issue an operation after it has handled the operation that caused the matrix creation.

The picture for the maximum number of concurrent matrices is less clear. Two factors influence this variable; it increases with the mean lifetime of matrix, but decreases with the operation rate as less matrices are generated. In total, a slight increase can be seen, which seems to be heavily dependent on internal dynamics of the system.

As expected, the increase of the operation rate does not show a significant change in the number of re-ordered operations because there is no increase in the potential for concurrency.

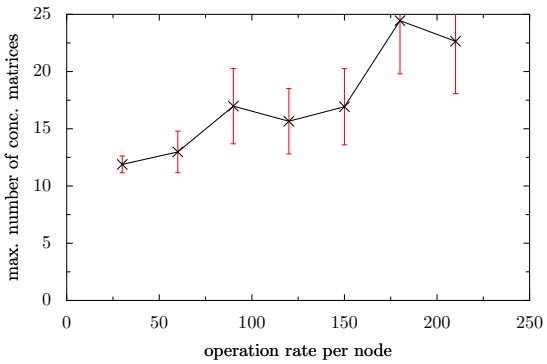
7.2.3 Influence of Link Latency

The effect of link latency is an important factor when a replication system has to scale beyond the local network. The following experiment is investigating the effect of an increasing round-trip time between hosts increases on the behavior of the DCC algorithm. To this end, we increase the link latency from a time that is typical for a LAN (we chose 1 ms) up to 80 ms, which is a network latency that is not untypical



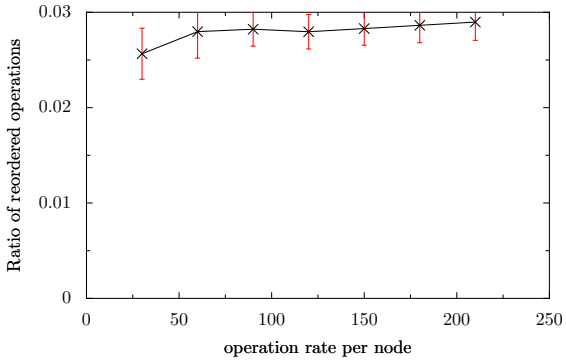
operation rate per node	30.0	60.0	90.0	120.0	150.0	180.0	210.0
mean lifet. of key matrices (ms)	111.0	55.93	36.79	27.26	21.38	17.70	15.24
± 95% confidence interval	5.188	4.049	2.220	1.670	1.342	1.189	1.328

Figure 7.5: Operation Rate Influence: Matrix Lifetime



operation rate per node	30.0	60.0	90.0	120.0	150.0	180.0	210.0
max. number of conc. matrices	11.88	12.97	16.97	15.65	16.92	24.45	22.64
± 95% confidence interval	0.736	1.817	3.290	2.855	3.327	4.649	4.576

Figure 7.6: Operation Rate Influence: Concurrently existing matrices



operation rate per node	30.0	60.0	90.0	120.0	150.0	180.0	210.0
Ratio of reordered operations	0.025	0.027	0.028	0.027	0.028	0.028	0.028
± 95% confidence interval	0.002	0.002	0.001	0.001	0.001	0.001	0.001

Figure 7.7: Operation Rate Influence: Reorder Probability

for WANs (such as for intercontinental connections)[2].

Experiment Setup We examined the DCC algorithm in a setup of replicas, each of which executed 110 operations per second, issued in batches of 3. During the experiment we increased the link latency from 1ms to 80 ms (1ms, 20ms, 40ms, 60ms, 80ms) and observed the number of concurrency matrices and the reorder probability for operations.

Results and Discussion In Fig. 7.8 we can observe that the mean lifetime of the matrices increases linearly with the link latency, as expected. The mean lifetime of the matrix corresponds to the mean time in which concurrency is possible, which should be approximately the round trip time between the two hosts. It increases from a life time of 29 ms for a round-trip time of 2 ms to 158 ms for a round-trip time of 160 ms.

Similarly, the increasing link latency has a linear affect on the maximum number of concurrently existing per-key matrices, because key matrices live longer (Fig. 7.9).

Just as the matrix lifetime, the re-order probability also increases linearly with the link latency (Fig. 7.10). A larger link latency results in a longer time window in which concurrency can happen and so the probability for a re-order to be necessary increases.

7.2.4 Influence of Size of the Hot Data Set

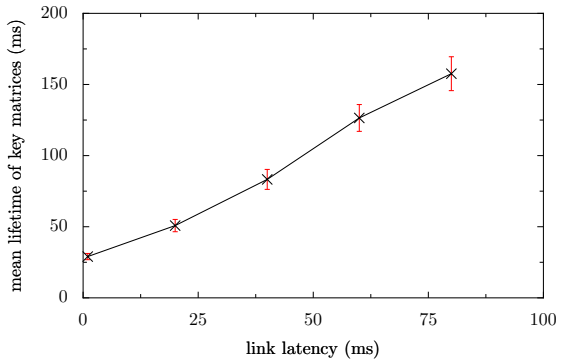
Due to its design, the behavior of DCC is not affected by the size of the replicated data set. However, the probability for conflicting changes should increases when writes concentrate on a small part of the data. The next experiment estimates this effect of the size of the data set under change on re-order probability and verifies that size of the dynamic state is independent of it.

Experiment Setup In this experiment, we set the link latency to 1ms and increased the size of the hot data set from 100 to 1000 rows (100, 250, 500, 750, 1000 rows). The operations changed rows of this set with an equal probability.

Results and Discussion Increasing the size of the hot data set from 100 to 1000 rows induces only a slight increase in matrix lifetime and the maximum number of matrices (Fig. 7.11 and Fig. 7.12). This is because matrices are created with a first write to a particular key and deleted shortly thereafter. When these writes are scattered over more keys, the potential for collision increases slightly. Then a write does not create a new matrix but updates an existing one.

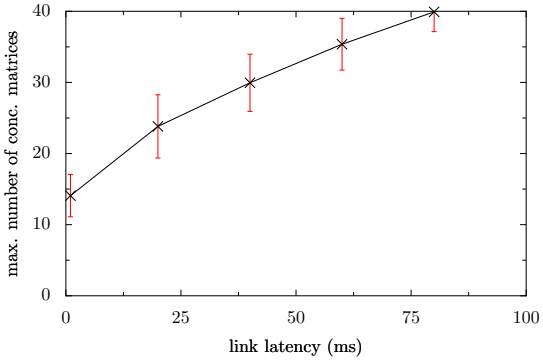
Fig. 7.13 shows that the re-order probability is inversely proportional to the size of the data set. This is explained by the fact that a larger data set has a smaller

[2]Note that this is the link latency, not the round-trip time as measured by ping.



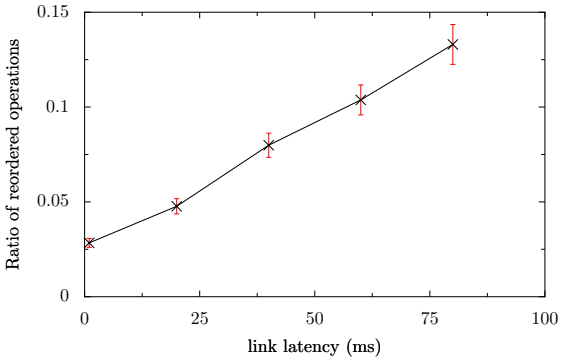
link latency (ms)	1.0	20.0	40.0	60.0	80.0
mean lifetime of key matrices (ms)	29.01	50.80	83.26	126.4	157.5
± 95% confidence interval	2.186	4.315	7.019	9.425	11.91

Figure 7.8: DCC: Link latency vs. matrix lifetime



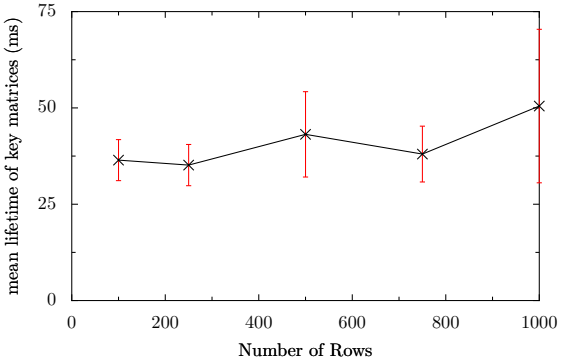
link latency (ms)	1.0	20.0	40.0	60.0	80.0
max. number of conc. matrices	14.07	23.81	29.94	35.36	39.92
± 95% confidence interval	2.970	4.445	4.019	3.633	2.780

Figure 7.9: DCC: Link latency vs. max. number of matrices



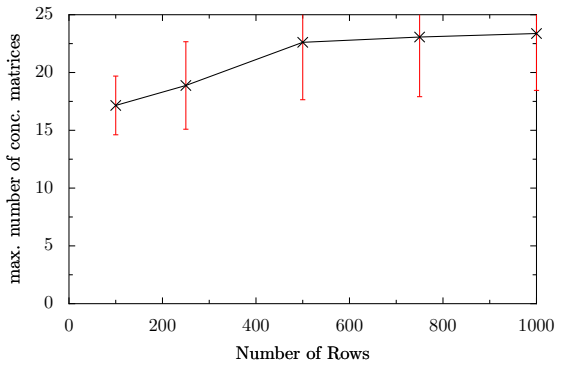
link latency (ms)	1.0	20.0	40.0	60.0	80.0
Ratio of reordered operations	0.028	0.047	0.079	0.103	0.133
± 95% confidence interval	0.002	0.004	0.006	0.007	0.010

Figure 7.10: DCC: Link latency vs. reorder probability



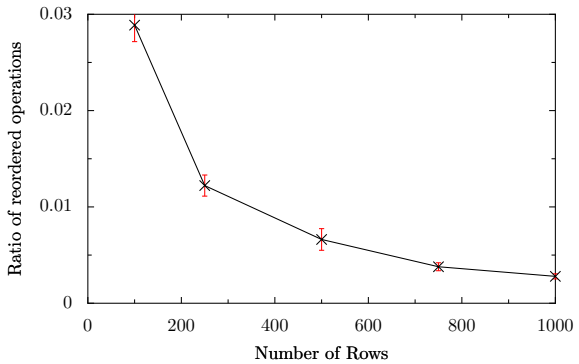
Number of Rows	100.0	250.0	500.0	750.0	1000.
mean lifetime of key matrices (ms)	36.44	35.16	43.14	38.02	50.48
± 95% confidence interval	5.325	5.350	11.07	7.243	19.92

Figure 7.11: DCC: Data Size vs. Matrix Lifetime



Number of Rows	100.0	250.0	500.0	750.0	1000.
max. number of conc. matrices	17.15	18.87	22.61	23.06	23.37
± 95% confidence interval	2.539	3.784	4.966	5.161	4.921

Figure 7.12: DCC: Data Size vs. State Size



Number of Rows	100.0	250.0	500.0	750.0	1000.
Ratio of reordered operations	0.028	0.012	0.006	0.003	0.002
± 95% confidence interval	0.001	0.001	0.001	0.000	0.000

Figure 7.13: DCC: Data Size vs. Reorder Probability

probability that two writes change the same row within the time window in which concurrency is possible.

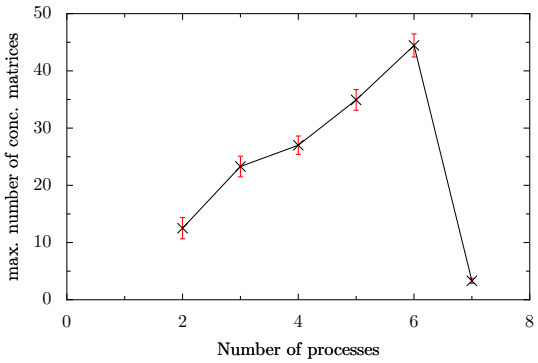
7.2.5 Influence of Number of Replicas

In our last experiments we attempted to assess the scalability of our algorithm in terms of number of replicas.

Experiment Setup We conducted two experiments, one with a constant operation rate per node, the other with a constant total operation rate. In the first experiment, the increase in the number of replicas means an increase in the total operation rate that has to be handled by each node. We scaled the number of replicas from 2 to 6 nodes and maintained a constant issuing rate of 100 ops/s at each replica. In the second experiment, the operation rate at each replica was constant. We increased the number of replicas from 2 to 7 nodes while decreasing the issuing operation rate from 100 ops/s to 28 ops/s, resulting in about 200 ops/s to be handled by each replica.

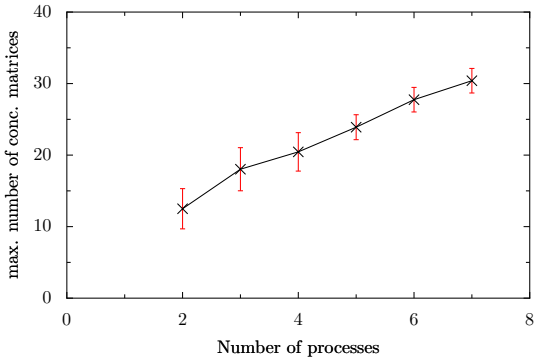
Results and Discussion Fig. 7.14 shows a linear increase in the maximum number of per-key matrices. With 6 replicas, i.e. 600 ops/s handled at each replica, the system becomes overloaded. Because our implementation does not contain proper scheduling and back-pressure on the load generator, the system practically ceases to operate.

Fig. 7.15 also shows a linear increase in the maximum number of per-key matrices, but has a smaller slope than the previous curve. This increase indicates a larger potential for causality, which can be explained by the fact that in order to remove a matrix acknowledgments from all replicas need to be received.



Number of processes	2.0	3.0	4.0	5.0	6.0	7.0
max. number of conc. matrices	12.50	23.28	27.00	34.92	44.44	3.307
± 95% confidence interval	1.858	1.806	1.608	1.808	2.018	0.446

Figure 7.14: DCC: No. replicas vs. State Size (variable rate)



Number of processes	2.0	3.0	4.0	5.0	6.0	7.0
max. number of conc. matrices	12.5	18.02	20.45	23.90	27.74	30.40
± 95% confidence interval	2.811	3.014	2.687	1.743	1.717	1.717

Figure 7.15: DCC: No. replicas vs. State Size (constant rate)

Chapter 8

Conclusion

This dissertation contributes a solution to the problem of designing and implementing weak-consistency replication systems. Any solution to this problem has to take care that it provides a convenient programming model, maintains the theoretical advantages of a weak-consistency architecture and facilitates a lightweight and efficient implementation.

As a solution to the replication problem, we propose a framework of efficient and lightweight mechanisms, algorithms and protocols, which is based on the distributed systems concepts of **causality and causal consistency**. Causality is inherent in any communication in a distributed system and causal consistency provides the programmer with a familiar memory model. In developing our framework, we follow a **bottom-up approach** that leads from change dissemination and persistence mechanisms to high-level abstractions for ensuring consistency in face of concurrent changes. By using the bottom-up approach that stays close to available system primitives for message communication and sequential log access, we ensure that our framework is efficient in practice and results in a lightweight implementation. This process leads to replication mechanisms that are not designed towards a specific use case, but guarantee efficiency for all applications with specifications that match the provided programming model.

The replication framework of this dissertation contains two protocols that provide change dissemination for causal consistency. The **causal gossip** protocol is an efficient way to track and communicate changes and their causal dependency for periodic reconciliation of replicas in mobile environments. The **direct send** protocol is a reliable multicast protocol for disseminating changes between online replicas. Together with our change tracking semantics it provides high-performance non-blocking FIFO delivery of changes, which ensures high availability of data and a loose coupling between replicas, and is able to support higher-level protocols in delivering causal consistency guarantees.

Our decision for causal consistency is complemented by the choice of a **persistent log** as the underlying storage structure. We showed how the log's append-only write interface can be tightly integrated with our causality-based mechanisms and protocols. By strictly using the log in an append-only manner we can ensure high-

performance persistence for our system. We introduced the concept of replayability as well as mechanisms that enforce a logical order for log entries that is different the from their physical order. With this foundation we are able to use the persistent replication log also as a database redo log. This unification of database and replication log allows us to guarantee both local and remote consistency of the replicated data in a simple manner.

In order to be able to make causal consistency usable for replication, we augmented causal consistency guarantees by two suites of lightweight algorithms. These algorithms provide causal consistency for non-concurrent changes and handle concurrent changes consistently on all replicas based on the novel concept of **interpreting concurrent changes as branches**. The **Distributed Consistent Branching** algorithm explicitly recreates branches on all replicas. It can be readily applied in off-line replication systems where branches may be explicitly created and later merged manually or by automatic conflict resolvers. The **Distributed Consistent Cutting** algorithm selects one branch consistently and thereby enforces total causal order for its changes. DCC can be applied to on-line or off-line replication systems that require causal consistency and provides a light-weight programming model that can act as design alternative in the range between basic replication algorithms such as Last-Writer-Wins (LWW) and strong, sequential consistency.

All mechanisms were **implemented as part of the LogDB database system** in order to verify their suitability for a real database system. The database system has been put under generated load while observing some key characteristics of our algorithms. These experiments give evidence that our contributions can be applied to real-world replication problems and allow the reader to assess the behavior of our algorithms under varying conditions.

Bibliography

[1] S. Abiteboul, R. Agrawal, P. Bernstein, M. Carey, S. Ceri, B. Croft, D. DeWitt, M. Franklin, H. G. Molina, D. Gawlick, J. Gray, L. Haas, A. Halevy, J. Hellerstein, Y. Ioannidis, M. Kersten, M. Pazzani, M. Lesk, D. Maier, J. Naughton, H. Schek, T. Sellis, A. Silberschatz, M. Stonebraker, R. Snodgrass, J. Ullman, G. Weikum, J. Widom, and S. Zdonik. The Lowell database research self-assessment. *Communications of the ACM*, 48(5):111–118, 2005. ISSN 0001-0782.

[2] N. Adly. HARP: a hierarchical asynchronous replication protocol for massively replicated systems. Technical Report UCAM-CL-TR-310, University of Cambridge, Computer Laboratory, Aug. 1993.

[3] D. Agrawal, A. E. Abbadi, and R. C. Steinke. Epidemic algorithms in replicated databases (extended abstract). In *PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 161–172, New York, NY, USA, 1997. ACM. ISBN 0-89791-910-6.

[4] R. Agrawal, A. Ailamaki, P. A. Bernstein, E. A. Brewer, M. J. Carey, S. Chaudhuri, A. Doan, D. Florescu, M. J. Franklin, H. Garcia-Molina, J. Gehrke, L. Gruenwald, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. F. Korth, D. Kossmann, S. Madden, R. Magoulas, B. C. Ooi, T. O'Reilly, R. Ramakrishnan, S. Sarawagi, M. Stonebraker, A. S. Szalay, and G. Weikum. The Claremont report on database research, May 2008.

[5] S. Alagar and S. Venkatesan. Causal ordering in distributed mobile systems. *IEEE Trans. Comput.*, 46(3):353–361, 1997. ISSN 0018-9340.

[6] P. S. Almeida, C. Baquero, and V. Fonte. Version stamps – decentralized version vectors. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 544–551. IEEE Computer Society, 2002. ISBN 0-7695-1585-1.

[7] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, and causal. In *IEEE Transactions on Software Engineering*, pages 229–236, 1995.

[8] L. Alvisi, K. Bhatia, and K. Marzullo. Causality tracking in causal message-logging protocols. *Distributed Computing*, 15(1):1–15, 2002. ISSN 0178-2770.

[9] R. Baldoni, R. Guerraoui, R. R. Levy, V. Quéma, and S. Tucci Piergiovanni. Unconscious Eventual Consistency with Gossips. In *Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006)*, pages 65–81, 2006.

[10] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, May 2006.

[11] N. Belarmani, M. Dahlin, A. Nayate, and J. Zhen. Making replication simple with Ursa. Technical Report TR-07-57, University of Texas at Austin Department of Computer Sciences, 2007.

[12] K. Bhatia, K. Marzullo, and L. Alvisi. Scalable causal message logging for wide-area environments. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 864–873, London, UK, 2001. Springer-Verlag. ISBN 3-540-42495-4.

[13] K. Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *SIGOPS Operating Systems Review*, 28(1):11–21, 1994. ISSN 0163-5980.

[14] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computing Systems*, 17(2):41–88, 1999. ISSN 0734-2071.

[15] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Deconstructing Paxos. *SIGACT News*, 34(1):47–67, 2003. ISSN 0163-5700.

[16] P. Brereton. Detection and resolution of inconsistencies among distributed replicates of files. *SIGOPS Oper. Syst. Rev.*, 17(1):10–15, 1983. ISSN 0163-5980.

[17] M. Burrows. Chubby distributed lock service. In *Proceedings of the 7^{th} Symposium on Operating System Design and Implementation, OSDI'06*, pages 335–350. USENIX Association, November 2006.

[18] C. Q. Castro and T. Pakkala. Version vectors based synchronization engine for mobile devices. In *PDCN'07: Proceedings of the 25th conference on Proceedings of the 25th IASTED International Multi-Conference*, pages 309–314, Anaheim, CA, USA, 2007. ACTA Press.

[19] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: the gaps between theory and practice. In J. T.-L. Wang, editor, *SIGMOD Conference*, pages 739–752. ACM, 2008. ISBN 978-1-60558-102-6.

[20] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-616-5.

[21] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[22] O. M. Cheiner and A. A. Shvartsman. Implementing and evaluating an eventually-serializable data service. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, page 317, New York, NY, USA, 1998. ACM. ISBN 0-89791-977-7.

[23] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. *SIGOPS Operating Systems Review*, 27(5): 44–57, 1993. ISSN 0163-5980.

[24] D. R. Cheriton and D. Skeen. Comments on the respones by Birman, van Renesse and Cooper. *SIGOPS Operating Systems Review*, 28(1):32, 1994.

[25] Chittajullu and Mcmillin. History clipping in event-driven distributed systems. In *Proceedings of the ISCA 13th International Conference on Parallel and Distributed Computing Systems*, pages 460–465, August 8-10 2000.

[26] R. Cooper. Experience with causally and totally ordered communication support: a cautionary tale. *SIGOPS Operating Systems Review*, 28(1):28–31, 1994. ISSN 0163-5980.

[27] R. Cox and W. Josephson. Communication timestamps for filesystem synchronization. Technical Report TR-01-01, Harvard Computer Science, 2001.

[28] R. Cox and W. Josephson. Optimistic replication using vector time pairs, 2004.

[29] R. Cox and W. Josephson. File synchronization with vector time pairs. Technical Report MIT-LCS-TM-650, MIT LCS, 2005.

[30] J. D S Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *Distributed systems, Vol. II: distributed data base systems*, pages 306–312, 1986.

[31] M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication for large-scale systems. Technical Report TR-04-28, The University of Texas at Austin, 2004.

[32] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007. ISSN 0163-5980.

[33] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004. ISSN 0360-0300.

[34] A. Demers, D. Greene, C. Houser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. *ACM SIGOPS Operating Systems Review*, 22(1):8–32, 1988. ISSN 0163-5980.

[35] A. R. Downing, I. B. Greenberg, and J. M. Peha. OSCAR: A system for weak-consistency replication. In *Workshop on the Management of Replicated Data*, pages 26–30, 1990.

[36] T. Ekenstam, C. Matheny, P. Reiher, and G. J. Popek. The Bengal database replication system. *Distributed Parallel Databases*, 9(3):187–210, 2001. ISSN 0926-8782.

[37] E. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers*, 41(5):526–531, 1992.

[38] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulie. From Epidemics to Distributed Computing. *IEEE Computer*, 37(5):60–67, 2004.

[39] K. Fall. A delay-tolerant network architecture for challenged internets. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–34, New York, NY, USA, 2003. ACM. ISBN 1-58113-735-4. doi: http://doi.acm.org/10.1145/863955.863960.

[40] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 300–309, New York, NY, USA, 1996. ACM. ISBN 0-89791-800-2.

[41] L. Gao, M. Dahlin, J. Zheng, L. Alvisi, and A. Iyengar. Dual-quorum replication for edge services. In *Middleware*, pages 184–204, Nov. 2005.

[42] L. Gao, A. Nayate, and J. Zheng. Improving availability and performance with application-specific data replication. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):106–120, 2005. ISSN 1041-4347.

[43] A. Gidenstam and M. Papatriantafilou. Adaptive plausible clocks. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 86–93, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2086-3.

[44] D. K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM. ISBN 0-89791-009-5.

[45] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. ISSN 0163-5700.

[46] D. Goldin and P. Wegner. The church-turing thesis: Breaking the myth. In *New Computational Paradigms*, volume 3526, pages 152–168. Springer Berlin, Heidelberg, 2005.

[47] R. A. Golding. Weak consistency group communication for wide-area systems. In *Workshop on the Management of Replicated Data*, pages 13–16, 1992.

[48] R. A. Golding. *Weak-consistency group communication and membership*. PhD thesis, University of California at Santa Cruz, Santa Cruz, CA, USA, 1992.

[49] R. A. Golding, D. D. E. Long, and J. Wilkes. The refdbms distributed bibliographic database system. In *1994 Winter USENIX*, pages 47–62, San Francisco, CA, January 17-21 1994. USENIX.

[50] J. Gray and M. Compton. A call to arms. *Queue*, 3(3):30–38, 2005. ISSN 1542-7730.

[51] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 173–182. ACM Press, 1996.

[52] M. B. Greenwald, S. Khanna, K. Kunal, B. C. Pierce, and A. Schmitt. Agreeing to agree: Conflict resolution for optimistically replicated data. In S. Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2006. ISBN 3-540-44624-9.

[53] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, Berlin, Germany, 2006.

[54] R. G. Guy, G. J. Popek, and T. W. P. Jr. Consistency algorithms for optimisic replication. In *Proceedings of the First International Conference on Network Protocols*, pages 250–261, 1993.

[55] R. G. Guy, P. L. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. In *ER '98: Proceedings of the Workshops on Data Warehousing and Data Mining*, pages 254–265, London, UK, 1999. Springer-Verlag. ISBN 3-540-65690-1.

[56] Y. Hamadi and M. Shapiro. Pushing log-based reconciliation. *International Journal on Artificial Intelligence Tools*, 14(3):445–458, 2005.

[57] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 981–992, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6.

[58] A. Heddaya, M. Hsu, and W. Weihl. Two phase gossip: managing distributed event histories. *Information Sciences*, 49(1-3):35–57, 1989. ISSN 0020-0255.

[59] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[60] J. Holliday, R. Steinke, D. Agrawal, and A. E. Abbadi. Epidemic algorithms for replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1218–1238, 2003. ISSN 1041-4347.

[61] Y.-W. Huang and P. S. Yu. Lightweight version vectors for pervasive computing devices. In *ICPP '00: Proceedings of the 2000 International Workshop on Parallel Processing*, page 43, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0771-9.

[62] F. Hupfeld. Log-structured storage for efficient weakly-connected replication. In *ICDCS Workshops*, pages 458–463, 2004.

[63] F. Hupfeld and M. Gordon. Using distributed consistent branching for efficient reconciliation of mobile workspaces. In *Proceedings of the 2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing (IEEE CollaborateCom) Workshops*, pages 1–9, 2006.

[64] F. Hupfeld, T. Cortes, B. Kolbeck, E. Focht, M. Hess, J. Malo, J. Marti, J. Stender, and E. Cesario. XtreemFS – a case for object-based file systems in grids. *Concurrency and Computation: Practice and Experience*, 20, 2008.

[65] F. Hupfeld, B. Kolbeck, J. Stender, M. Högqvist, T. Cortes, J. Marti, and J. Malo. FaTLease: Scalable fault-tolerant lease negotiation with Paxos. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 1–10, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-997-5.

[66] U. Irmak, S. Mihaylov, and T. Suel. Improved single-round protocols for remote file synchronization. In *INFOCOM*, pages 1665–1676, 2005.

[67] R. Jiménez-Peris, M. Patino-Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM Transactions on Database Systems*, 28(3):257–294, 2003. ISSN 0362-5915.

[68] T. Johnson and K. Jeong. Hierarchical matrix timestamps for scalable update propagation. Technical Report TR96-017, University of Florida, 1996.

[69] D. S. P. Jr., G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. A. Edwards, S. Kiser, and C. S. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Software Eng.*, 9(3):240–247, 1983.

[70] B. B. Kang, R. Wilensky, and J. Kubiatowicz. Hash history approach for reconciling mutual inconsistency in optimistic replication. In *23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03)*, pages 670–677, 2003.

[71] A. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The icecube approach to the reconciliation of divergent replicas. In *Proc. of Twentieth ACM Symposium on Principles of Distributed Computing PODC, Newport, RI USA*, pages 210–218, 2001.

[72] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 190–201, New York, NY, USA, 2000. ACM. ISBN 1-58113-317-0.

[73] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions Computing Systems*, 10(4):360–391, 1992. ISSN 0734-2071.

[74] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. ISSN 0001-0782.

[75] L. Lamport. On interprocess communication, part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[76] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1997. ISSN 0018-9340.

[77] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[78] L. Lamport. Paxos made simple. *SIGACT News*, 32(4):18–25, 2001.

[79] B. W. Lampson. How to build a highly available system using consensus. In *WDAG '96: Proceedings of the 10th International Workshop on Distributed Algorithms*, pages 1–17, London, UK, 1996. Springer-Verlag. ISBN 3-540-61769-8.

[80] D. Lomet. The case for log structuring in database systems. In *6th International Workshop on High Performance Transaction Systems (HPTS)*, 1995.

[81] D. Malkhi and D. B. Terry. Concise version vectors in WinFS. *Distributed Computing*, 20(3):209–219, 2007.

[82] K. Manassiev and C. Amza. Scalable database replication through dynamic multiversioning. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 141–154. IBM Press, 2005.

[83] L. W. Mcvoy. Extent-like performance from a unix file system. In *In Proceedings of the 1991 Winter USENIX Conference*, pages 33–43, 1991.

[84] Y. Minsky and A. Trachtenberg. Practical set reconciliation. Technical Report BU ECE–2002-01, Boston University, 2002.

[85] M. Mizuno, M. Raynal, and J. Z. Zhou. Sequential consistency in distributed systems: Theory and implementation. Technical Report RR-2437, INRIA, 1995.

[86] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17:94–162, 1992.

[87] A. Mostefaoui and M. Raynal. Causal multicasts in overlapping groups: Towards a low cost approach. Technical Report IRISA-PI - 93-710, Institut de recherche en informatique et systèmes aléatoires (IRISA), Rennes, FRANCE, 1993.

[88] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 174–187, New York, NY, USA, 2001. ACM. ISBN 1-58113-389-8.

[89] K. Nørvåg and K. Bratbergsengen. Log-only temporal object storage. In *Proceedings of the 8th International Workshop on Database and Expert Systems Applications, DEXA'97*, pages 728–733, 1997.

[90] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating SystemsPrinciples (SOSP-16), Saint Malo, France*, pages 288–301, 1997.

[91] R. Prakash, M. Raynal, and M. Singhal. An efficient causal ordering algorithm for mobile computing environments. In *ICDCS*, pages 744–751, 1996.

[92] R. Prakash, M. Raynal, and M. Singhal. An adaptive causal ordering algorithm suited to mobile computing environments. *Journal of Parallel and Distributed Computing*, 41(2):190–204, 1997.

[93] M. Rabinovich, N. H. Gehani, and A. Kononov. Scalable update propagation in epidemic replicated databases. In *EDBT '96: Proceedings of the 5th International Conference on Extending Database Technology*, pages 207–222, London, UK, 1996. Springer-Verlag. ISBN 3-540-61057-X.

[94] M. Raynal and M. Singhal. Logical time: Capturing causality in distributed systems. *Computer*, 29(2):49–56, 1996. ISSN 0018-9162.

[95] P. L. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the Ficus file system. In *USENIX Summer*, pages 183–195, 1994.

[96] K. Rimey. Version headers for flexible synchronization and conflict resolution. Technical Report 2004-3, Helsinki Institute for Information Technology, November 2004.

[97] L. Rodrigues and P. Verissimo. Causal separators and topological timestamping: an approach to support causal multicast in large-scale systems. Technical report, IST - INESC, Lisboa, Portugal, 1994.

[98] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1): 26–52, 1992.

[99] F. Ruget. Cheaper matrix clocks. In G. Tel and P. M. B. Vitányi, editors, *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG94)*, volume 857, pages 355–369, Terschelling, The Netherlands, 29 – 1 1994. Springer-Verlag. ISBN ISBN 3-540-58449-8.

[100] Y. Saito and H. M. Levy. Optimistic replication for internet data services. In *DISC '00: Proceedings of the 14th International Conference on Distributed Computing*, pages 297–314, London, UK, 2000. Springer-Verlag. ISBN 3-540-41143-7.

[101] Y. Saito and M. Shapiro. Optimistic replication. *Computing Surveys*, 37(1): 42–81, Mar. 2005.

[102] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, availability and performance in porcupine: a highly scalable, cluster-based mail service. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 1–15, New York, NY, USA, 1999. ACM. ISBN 1-58113-140-2.

[103] S. K. Sarin and N. A. Lynch. Discarding obsolete information in a replicated database system. *IEEE Transactions on Software Engineering*, 13(1):39–47, 1987. ISSN 0098-5589.

[104] M. Scholz, F. Bregulla, and A. Hinze. Using physical clocks for replication in manets. In *PERCOMW '07: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 114–119, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2788-4.

[105] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *Network Systems Design and Implementation (NSDI)*, pages 239–252, May 2006.

[106] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994. ISSN 0178-2770.

[107] M. Seltzer. Beyond relational databases. *Communications of the ACM*, 51(7): 52–58, 2008. ISSN 0001-0782.

[108] M. Shapiro, N. Preguiça, and J. O'Brien. Rufis: mobile data sharing using a generic constraint-oriented reconciler. In *2004 IEEE International Conference on Mobile Data Management (MDM'04)*, volume 0, page 146, Los Alamitos, CA, USA, 2004. IEEE Computer Society. ISBN 0-7695-2070-7.

[109] W. R. Stevens. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993. ISBN 0-201-63346-9.

[110] M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases (VLDB '07)*, pages 1150–1160, 2007.

[111] J. T. W. Page, R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on optimistically replicated, peer-to-peer filing. *Software–Practice and Experience*, 28(2):155–180, 1998. ISSN 0038-0644.

[112] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS '94: Proceedings of the third international conference on on Parallel and distributed information systems*, pages 140–150, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-6401-0.

[113] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected

replicated storage system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-715-4.

[114] D. B. Terry, K. Petersen, M. Spreitzer, and M. Theimer. The case for non-transparent replication: Examples from Bayou. *Data Engineering Bulletin*, 21 (4):12–20, 1998.

[115] The Open Group. The Single Unix Specification, Version 3, 2004.

[116] A. Traud, J. Nagler-Ihlein, F. Kargl, and M. Weber. Cyclic data synchronization through reusing SyncML. In *2008 IEEE International Conference on Mobile Data Management (MDM'08)*, pages 165–172, Los Alamitos, CA, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3154-0.

[117] R. van Renesse. Causal controversy at Le Mont St.-Michel. *SIGOPS Operating Systems Review*, 27(2):44–53, 1993. ISSN 0163-5980.

[118] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[119] W. Vogels, R. van Renesse, and K. Birman. The power of epidemics: robust communication for large-scale distributed systems. *SIGCOMM Computer Communication Review*, 33(1):131–135, 2003. ISSN 0146-4833.

[120] W. Wang, Y. Zhao, and R. Bunt. Hylog: A high performance approach to managing disk layout. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 144–158, 2004.

[121] G. T. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 233–242, New York, NY, USA, 1984. ACM. ISBN 0-89791-143-1.

[122] H. Yu and A. Vahdat. Combining generality and practicality in a conit-based continuous consistency model for wide-area replication. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 429, Washington, DC, USA, 2001. IEEE Computer Society.

[123] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 29–42, New York, NY, USA, 2001. ACM. ISBN 1-58113-389-8.

[124] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3): 239–282, 2002. ISSN 0734-2071.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Dissertationsschrift

"Causal Weak-Consistency Replication - A Systems Approach"

selbständig und ohne unerlaubte Hilfe angefertigt habe, und erkläre,

dass ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe noch einen solchen besitze,

dass mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt Universität zu Berlin (Amtliches Mitteilungsblatt der Humboldt-Universität Nr. 34/2006) bekannt ist.

Berlin, den 15.10.2008

Felix Hupfeld